A VERIFICATION APPROACH TO OBJECT-ORIENTED REQUIREMENTS SPECIFICATION IN SOFTWARE DEVELOPMENT FOR DISTRIBUTED COMPUTING SYSTEMS

By

KEUNHYUK YEOM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1995

To my mother, wife and son

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor and chairman, Professor Stephen S. Yau, for his guidance and encouragement throughout this research. During my graduate study, he has made contributions in various ways; offering, advice, insight, alternative solutions and encouragement. I also want to thank Professor Paul A. Fishwick, Professor Li-Min Fu, Professor Andrew F. Laine and Professor Paul Chun for their participation on my committee. Each provided me with helpful comments and good advice.

Additionally, I would like to thank all my colleagues who gave me encouragement and assistance during my studies here. I have enjoyed working with all of them. I also would like to acknowledge the financial and technical support from the Hitachi Co., Japan.

Finally, no words can adequately describe the importance of the support and encouragement from my mother, wife and brothers. Without their love and patience, this work would surely not have been completed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF NOTATIONS

Symbol	Description
S_b	: a distributed computing system unit.
S_{ads}	: an autonomous decentralized system.
$Spec[Sys]$	: a requirements specification of a system.
Sys	: a software system.
$Spec[C_i]$	: a requirements specification of a class.
C_i	: a class.
I_i	: a superclass.
A_i	: an attribute of a class.
M_i	: a method of a class.
T_i	: a state transition based on the dynamic behavior showing the events the object receives and sends.
S_N	: a next state.
S_C	: a current state.
I	: an input.
O	: an output.
$\longmapsto$	: a map to.
$\times$	: a cartesian product.
F	: a function.
G	: a function.

$\Box$	: always.
$\rightarrow$	: an implication.
$\Rightarrow$	: an entailment (a stronger type of implication).
$\neg$	: a negation.
$\leq$	: less than.

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

A VERIFICATION APPROACH TO OBJECT-ORIENTED REQUIREMENTS SPECIFICATION IN SOFTWARE DEVELOPMENT FOR DISTRIBUTED COMPUTING SYSTEMS

By

Keunhyuk Yeom

August 1995

Chairman: Stephen S. Yau
Major Department: Computer and Information Science and Engineering

Developing software for distributed computing systems is challenging due to a lack of effective methodologies and tools. In particular, because many errors in the source code can be traced to errors in the requirements specification, it is especially important to have effective verification techniques for this crucial stage. In this dissertation, an approach to verification of object-oriented requirements specification (OORS) in software development for distributed computing systems is presented. A formal object-oriented requirements specification language is developed in which the object-oriented concepts have been integrated to the formal state transition specification without sacrificing the advantages of either. User-friendly graphical notations are introduced to incorporate formal methods in the software development process. In this approach, the requirements specification generated by object-oriented analysis is described using a formal specification language, which is transformed into graphical notations. Then, the completeness and consistency of the requirements

specification expressed in terms of the graphical notations is verified by comparing it with the original requirements statement.

CHAPTER 1
INTRODUCTION

As VLSI and communication technologies advance, distributed computing systems become more cost effective for various applications. Developing software for distributed computing systems is more challenging than for centralized computing systems due to the additional complications of interprocessor communication, synchronization, etc.[1, 2]. A distributed computing system can be considered as a number of hardware processors with each processor having certain computing capability, and a number of communication links among processors. From a software point of view, a distributed software system can be considered as a set of sequential processes with each process residing on a hardware processor. In turn, these processes cooperate with one another to fulfill a system-wide task. Although the development of these individual processes exhibits many of the common characteristics shared by the development of sequential software for centralized computing systems, the following characteristics of distributed system software are distinct and must be addressed by any software development methodology, specially for such systems.

- The processes in a distributed software system can be executed in parallel.
- The processes in a distributed software system are able to cooperate with one another to fulfill a system-wide task. The cooperation involves interprocess communication and synchronization.
- Software components can be mapped into hardware resources by partitioning and allocating software components based on the structure and performance requirements of the system.

Furthermore, distributed computing systems have the following major advantages over centralized systems: 1) Their performance can be greatly improved by increasing the parallel processing capability. 2) They can be made more fault tolerant due to their distributed nature. 3) The system can be easily expanded.

Object-oriented software development [3, 4, 5, 6, 7, 8] is rapidly gaining popularity. One reason for this is that the concept of classes and objects and the organization of objects naturally reflect the structure of software systems, especially distributed computing applications. An object-oriented approach encourages software engineers to work and think in terms of the application domain through most of the software development cycle. Its greatest benefits come from helping developers express abstract concepts clearly and communicate with customers.

Object-oriented analysis (OOA) [8, 9, 10] is the process of generating an object-oriented requirements specification (OORS) from the requirements statements in a natural language to support object-oriented software development. It starts from requirements statements and is concerned with devising a detailed, concise, understandable, and valid model of the real-world. The successful analysis model states what must be done, without restricting how it is done, and thereby avoids implementation decisions. The goal of this kind of analysis is to understand the problem as a preparation for design.

Since distributed computing systems are used in a wide variety of critical applications, these systems must be reliable. Insuring this reliability has become a very important problem. Verification is an essential part in the overall software life cycle. It is closely tied to the individual developmental steps. Verification is defined [11] as the comparison at each stage of the software life cycle to determine that there has been a correct translation of that stage into the next one. Each verification activity applies techniques which have proved effective in detecting errors

commonly made in that step of the development process. After the requirements specification has been established, verification must check its adequacy in fulfilling what is stated in the requirements statements. After the design phase, verification addresses the adequacy of the software design in meeting the software requirements specification. Likewise, after coding, verification checks the compliance of the coded software with the design. The verification executed in each phase of the software development must assure that the software requirements specification is translated in the design expressed in the software design description and further in the code. This should include compliance with any standards or codes of practice which have been adopted [11]. The overall verification process determines whether the software behaves in accordance with its requirements specification. The requirements verification ensures that the software is ready to proceed into design with a high degree of confidence. Verification of the requirements specification against the requirements statements is necessary to avoid an incorrect interpretation of the requirements statements by the system designers. Such an eventuality would be dangerous because it could lead to common mode failure in the final system. Considering that many errors found in the source code stem from inconsistent or incomplete requirements specification, the requirements verification is very important in developing reliable software for a distributed computing environment [12].

In this dissertation, a verification approach to object-oriented requirements specification in software development for distributed computing systems is presented. The approach is based on an integration of formal specification methods and graphical representations. Formal specification techniques are generally beneficial because a formal language makes specifications more concise and explicit. These techniques help us acquire greater insights into the system design, dispel ambiguities, and

maintain abstraction levels. The user-friendly graphical notations are widely accepted by practitioners. Hence, viable strategies for incorporating formal methods in the software development process are needed to facilitate a better understanding of these methods in real-world software development.

The dissertation is organized as follows. Chapter 2 presents background of this research, such as object-oriented analysis techniques, distributed computing systems, software development framework, and related works. Chapter 3 presents overall steps of our approach to the verification of object-oriented requirements specification in software development for distributed computing systems. Chapter 4 presents an overview of the formal object-oriented specification language. First, its syntax and semantics are briefly summarized; then, characteristics of the object-oriented and state transition specification are explained. Second, the procedure for producing a formal object-oriented specification from the object model [8], which provides the static structure among classes, and the dynamic model [8], which represents the behavior of the system, are provided. Chapter 5 presents the structures of each graphical notation and processes for building the graphical notations from the formal specification. Each graphical notation represents a different aspect of a system. An inheritance graph describes the inheritance relationships among classes. An information tree describes the static characteristics of classes and relationships among classes, such as associations, aggregations, and method invocations. A transition trace table represents all dynamic behavior of a system with an ordered list of transitions between different objects assigned to columns in a table. Chapter 6 explains a verification technique for checking the consistency and completeness between requirements statements and object-oriented requirements specification described by object and dynamic models are explained. Chapter 7 illustrates our requirements verification approach with an automated teller machine (ATM) system

example. Finally, Chapter 8 contains some concluding remarks and recommendations for future research.

CHAPTER 2
BACKGROUND

2.1 Object-Oriented Analysis Techniques

Object-Oriented Analysis (OOA) is rapidly gaining in popularity, promising to provide a more understandable specification and better support for object-oriented design and implementation. A systematic approach to OOA includes static modeling to specify the objects, classes, and their various relationships; dynamic modeling to specify class states, state transitions, class behavior and object interaction; managing specification complexity through the use of views or subjects; and the inclusion of system constraints and error handling.

In this section, the existing object-oriented analysis and design methodologies will be summarized.

2.1.1 Booch's Approach

Booch diagrams [3, 13, 14] are the earliest attempt to model object-oriented development graphically. The work was originally designed for implementation using Ada [15] as the programming language. Hence a significant portion of the notation is implementation specific and closely related to the features of Ada. Users of Booch diagrams may find themselves using an Ada tool rather than an object-oriented tool. Object-oriented concepts are not covered by Booch diagrams completely. For example, overloading and polymorphism are defined in an implicit way. Generic classes exist only as a notation and there is no recommendation concerning their instantiation.

Figure 2.1 gives an overview of the Booch process. Booch acknowledges that analysis and design cannot proceed in isolation, and advocates a piecemeal approach;

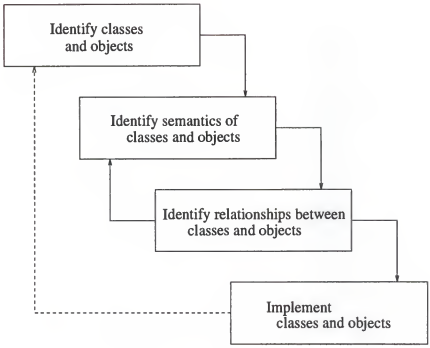

Figure 2.1. Booch: Process overview.

which allows us to keep improving the design until we are satisfied. Booch calls this "round trip gestalt design" and suggests this is the foundation of object-oriented development. Thus the Booch process is descriptive (saying what we *may* do) rather than prescriptive (saying what we *should* do) [16].

The first step is to identify the objects and classes that form part of the system. Then the interface for classes is constructed (this is "identifying the semantics") followed by finding relationships between the classes. The discovery of relationships may cause new interfaces to be added, so these two steps are iterated until a satisfactory state is obtained. The implementation stage decides on the representation of class internals (attributes and behavior); this in turn may result in having to apply the entire process to the behavior of a single class.

2.1.2 Object Modeling Technique

Object Modeling Technique (OMT) is proposed by Rumbaugh *et al.* [8, 17]. Figure 2.2 presents an overview of the OMT process. It is divided into three phases: analysis, which is concerned with modeling the real world; design, which decides on subsystems and overall architecture; and implementation, which encodes the design in a programming language. OMT suggests that an object-oriented system should be analyzed and designed using three models. Extended entity-relationship diagrams are used for the object model, state-transition diagrams for the dynamic model, and data flow diagrams for the functional model. It is known to have the most user-friendly notations and concepts among various object-oriented methods. The notations are sufficiently general so that they can be used easily by experienced users of the structured methodologies. Implementation techniques using object-oriented languages, non-object-oriented languages as well as relational databases have been proposed [8].

The analysis process concentrates on modeling the world. This method gives advice on identifying appropriate classes and associations between them. Attributes are added as necessary. The object model is refined using inheritance. The dynamic model is built by considering the events coming into the system and its responses; these are formalized with the state diagram. The functional model is then built by examining input and output values which are parameters of events and showing dependencies using a data-flow diagram; this process may uncover additional, internal, values.

As with any realistic development process, OMT analysis is iterative. The functional model can reveal extra objects and methods that have to be incorporated into the other two models. All three models should be compared with the requirements.

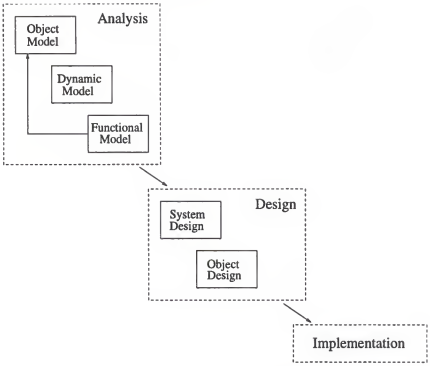

Figure 2.2. Object Modeling Technique: Process overview.

The analysis phase is completed when all the models are consistent and capture all the requirements.

2.1.3 Coad and Yourdon's OOA

Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) have been developed by Coad and Yourdon [10, 18]. The OOA model is divided into 5 layers: subjects, objects, structure, attributes and services. The subject layer controls the part of the model visualized by a user at a certain point in time. The object layer is represented by an Object Diagram, where a box is used to specify an object and the set of related attributes. Details of the attributes are given in the attribute layer in the form of a data dictionary. The structure layer specifies the relationships among objects, including generalization/specialization relationships (inheritance), whole/part relationships and instance connections. The service layer defines the

methods in the objects. All the layers are conglomerated into an "object diagram", since Coad and Yourdon propose that it is easier for users to learn only one type of diagram. However, their object diagram is in fact an overlay of data flow diagrams and extended entity-relationship diagrams, and may lead the user into unnecessary and confusing details in large systems [19].

2.1.4 Shlaer and Mellor's OOA

The Shlaer and Mellor technique [9] is based on data aspects of the problem. As presented [9], the OOA method is described in three steps: information models, state models, and process models.

Information Models

This step is a form of entity relationship (ER) modeling [20] common to many structured methods. It concerns identifying the entities relevant to a problem, modeled as objects, whose characteristics are described by attributes. Associations between objects are described by relationships which are formalized by *referential attributes.*

Objects, attributes, and relationships are depicted graphically on an **Information Structure Diagram** (ISD). Accompanying textual descriptions are provided in the equivalent of a data dictionary.

State Models

The second step involves describing the dynamic behavior of objects and relationships over time. For objects with dynamic behavior (that is, active objects), a **State Transition Diagram** (STD) is produced.

Transitions between behavior *states* are caused by the receipt of *events* which may represent external inputs or communication from another active object. The

entirety of event communication between objects in a model is depicted on an **Object Communication Diagram** (OCM). Note that all event communication in OOA is asynchronous.

Actions describe the activity or processing associated with completing a transition to a state, e.g. generating events to other objects or entities external to the model. Actions therefore embody the functionality of the model (within the STD framework).

Note that to disambiguate information state from behavior state we refer to this complete second step as **state transition modeling**.

Process Models

The final step of the method concerns refining the action descriptions of the previous step. The aim is to identify fundamental, reusable processes (each associated with a particular object) that can be combined to produce the complete effects of the actions.

Complex actions are represented graphically by a form of data flow diagram, the **Action Data Flow Diagram** (ADFD). ADFDs are used to identify the processes required by actions and how the processes may be combined. Non-trivial processes are given a process description, e.g. a piece of pseudo-code.

The **Object Access Model** (OAM) is a graphical representation of the processes belonging to objects which are invoked by other objects. It is akin to the OCM except that the OAM depicts synchronous rather than asynchronous communication.

2.1.5 Object-Oriented Structured Design

Object-Oriented Structured Design (OOSD) is a graphical tool proposed by Wasserman and coworkers [21, 22, 23] and is a combination of Booch diagrams [3],

Buhr diagrams [24] and Yourdon structure charts [25]. OOSD supports most of the object-oriented concepts. It is independent of any language, but can be augmented with language features, such as Ada guards, to facilitate implementation. A use for this tool in analysis and design has yet to be developed.

2.1.6 Others

Data flow diagrams [26, 27] and structure charts [28, 29] are standard graphic tools for structured analysis and design. Ward and Mellor [30] and Yourdon [25] have modified the data flow diagrams by adding control couples for real-time applications. They have also added entity-relationship diagrams [31] and state-transition diagrams [32]. Additionally, Jackson System Development (JSD) [33, 34] and Structured Systems Analysis and Design Methodology (SSADM) [35, 36] are other schools in structured analysis and design more popular in the UK. JSD emphasizes entity structures rather than data flows in the initial analysis phase, whereas SSADM emphasizes both. In spite of differences in emphases and symbols, the essence behind the graphical notations are quite similar to those of the Yourdon school.

There have been structured analysis methodologies already proposed [37, 38, 39] that have included object-oriented considerations. But, most of these object-oriented properties are largely implicit and obscure at best, and appear to be afterthoughts.

Other proposals include General Object-Oriented Software Development (GOOD) [40], Hierarchical Object-Oriented Design (HOOD) [41, 42], Object-Oriented System Analysis (OOSA) [43], and the notations of Henderson-Sellers [19] and Wirfs-Brock *et al.* [44].

2.2 Our Software Development Framework for Autonomous Decentralized System

Owing to the rapid development of computer and communication technologies, distributed computing systems are used widely in various areas of applications [1]. Most of those applications are directly related to our dynamic and expanding human society. There is also an emerging demand for a computing system based on newly developing technologies. In some application areas, it is essential that the system operates continuously in spite of partial system failures, expansions and maintenance [45]. In order to satisfy these requirements, the computing systems must be reliable, adaptable, and expandable. Therefore, these kinds of distributed systems should not only have the fault-tolerance capability, but also on-line expansion and on-line maintenance properties. However, it is difficult to attain these on-line properties in a distributed computing system through conventional techniques, which were developed for the centralized system concept. In order to have these properties, the Autonomous Decentralized System (ADS), which is composed of largely autonomous and decentralized components with the capabilities of on-line expansion, fault-tolerance, and on-line maintenance, has been developed [46, 47]. ADSs have been applied in several areas, such as the steel production control systems and train traffic control. In order to effectively utilize ADSs, effective application software development methodologies are needed [4, 6]. Because the object-oriented paradigm reflects the distributed structure of the problem space and is suitable for representing inherently concurrent behavior, it is suitable to support the software development for large scale distributed computing systems, like ADSs. We have developed an object-oriented computation model [4] and software development framework [6] for ADSs based on the computation model.

In this section, I will present an object-oriented software development framework for ADSs based on an object-oriented computation model. Our framework consists of object-oriented requirements analysis, system design, implementation, allocation, verification and maintenance.

2.2.1 Overview of ADS

In this section, I will give an overview of the ADS including its conceptual foundations, software structure, etc.

ADS Concept

The ADS concept is based on a biological analogy and incorporates the view that a system almost always has faulty parts and undergoes modifications [45, 48, 49]. This allows for the presence of partially damaged components in the system. Ihara and Mori [46] defined it mathematically utilizing concepts from control theory.

Definition 2.2.1 Subsystem
A subsystem S_b is a distributed computing system unit which includes a separate hardware processor with its own system software and application software in that processor.

The ADS S_{ads} consists of a set of subsystems, i.e., $S_b \in S_{ads}$. Each subsystem serves to control some part of the whole system's functions. This entails that the whole system's functions are performed by the integration of each distributed subsystem's operation. Each subsystem has responsibility for controlling its own computing process.

Definition 2.2.2 Autonomous controllability
The system has autonomous controllability if the following condition is satisfied: If any subsystem fails, the other subsystems can continue to manage themselves.

Autonomous controllability means that each subsystem is self-controllable and can control its own area without regard to other subsystems' conditions.

Definition 2.2.3 Autonomous coordinability
The system has autonomous coordinability if the following condition is satisfied: If any subsystem fails, the other subsystems can coordinate their individual objectives among themselves.

This states that a system's reliability is kept by process continuity when any subsystem fails, i.e., the other subsystem can coordinate their individual objectives among themselves.

Definition 2.2.4 Autonomous Decentralized System
A system is called an autonomous decentralized system if it has two autonomous properties — autonomous controllability and autonomous coordinability.

An ADS system consists of a set of distributed software subsystems. The software subsystem includes system software modules (subsystem software) and application software modules.

In order to constitute an autonomous decentralized software system, each subsystem software residing in a distributed processor is required to satisfy the following conditions [48]:

a) uniformity

b) equality

c) locality

In the ADS, each subsystem software must be uniform. The subsystem software can manage its own application software modules without being dependent

on the other subsystem software functions. Even if the subsystem software in some processors stop operation, the other subsystems' software can continue operation.

There is no master-slave relation among the software of the subsystems. Every subsystem software can manage its own application software modules without being directed by or giving directions to the other subsystem software. This condition assures that each subsystem software can continue its operation even if the other subsystem software functions fail. Each subsystem software must be able to manage its own application software and itself, as well as coordinate with other subsystems' software based only on local information.

ADS Software Structure

It is difficult to satisfy the above three conditions based on the conventional software structure and its management technique. On the other hand, with the autonomous decentralized software structure [48], these three conditions are successfully satisfied.

The basic feature introduced in the Autonomous Decentralized Software Structure is the *Data Field* (DF) where the data circulate among the modules in the software subsystem [46]. In the DF, each message has a content code that indicates its meaning. The software module broadcasts a message with the content code into the DF, and then determines whether or not to receive the message on the basis of the content code. The communication between software modules is locationally transparent and one-way directional. Any module can communicate with any other module connected to the DF. Every software module in the subsystem software and the application software is a functional unit that receives data from the DF and sends out data into the DF.

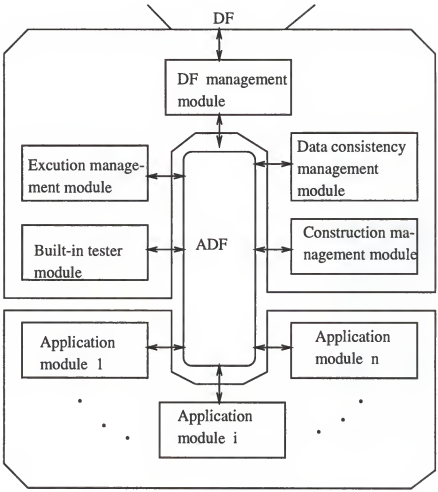

Figure 2.3. Modular software structure in a software subsystem.
(Taken from Mori et al [48] page 1058)

The subsystem software consists of the following modules: a DF management module, an execution management module, a built-in tester module, a construction management module and a data consistency management module. The DF management module receives messages from the DF or sends (broadcasts) messages to the DF. It updates a table that includes the relationship between the application software module in the software subsystem and the content codes necessary to execute the application software module. The execution management module monitors

the table. This table is called the atom data field (ADF). An atom consists a combination of subsystem software modules and application software modules in each processor. As soon as all the necessary data for the application software module are received in the table by the DF management module, the execution management module drives the application software module. The built-in tester module tests application software. The construction management module supports application software loading and expansion. The data consistency management module checks the replicated data received from duplicated modules and selects the correct one. This data consistency management module also supports the fault-tolerance of the ADS. The application software of an ADS consists of modules that execute their tasks independently. Each application module is controlled by the system software modules that reside in the same processor. In Figure 2.3, an example of the modular structure of each software subsystem for an ADS is shown.

2.2.2 Object-Oriented Computation Model

Before I present our software development framework, I will briefly summarize the computation model [4] which forms its basis.

The computation model is based on the object-oriented concept and supports on-line expandability and on-line modifiability at the application software level. In the computation model, ADS software is represented as a set of modules. Each module has its own control thread, object base, interface base, and base management mechanism (BMM). An object base contains instances of object classes defined in the module. The objects in the object base of a module are called the local objects of this module, and while those in the object bases of other modules are called the remote objects of this module. An interface base contains the object methods and the object names to which these methods belong. There are two types of

interface: an import interface and an export interface. The import interface is a list of methods in other objects which it invokes. The export interface is a list of own methods invoked by other objects. The BMM provides functions used to modify the object base and interface base dynamically to support on-line features.

The following are major features of the computation model:

- classes and objects used as bases of software development
- a two level encapsulation: module level and object level
- a guard statement for synchronization between modules
- an asynchronous remote object method invocation and synchronous local object method invocation
- a preservation of serialization in remote method invocations
- asynchronous remote communication control information in the content of communication
- full logical location transparency achieved by the location mechanism to determine the local object method invocation and the remote object method invocation
- exception handling to support retry, ignore, wait and time-out
- on-line expandability and on-line modifiability achieved by the BMM to modify the object base and interface base dynamically

2.2.3 Our Framework

Our framework has the following phases [6]: object-oriented requirements analysis, system design, implementation, allocation, verification and maintenance. The framework for software development for the ADS is shown in Figure 2.4.

We start the ADS software development with a set of requirements statements, which is transformed into the object model and dynamic model using object-oriented requirements analysis (OORA) technique. Our OORA technique is an extension of Object Modeling Technique (OMT) [8]. The OMT is based on constructing two visual projections of the system, called the object model and the dynamic model. The object model shows classes and their structures derived from the knowledge about application domain and the requirement statements. The dynamic model shows a finite-state machine model for each class in the object model. The object model and dynamic model are represented by the module description in Design Description Language (DDL) [4] in the system design phase. Then, each module design in DDL is implemented in C++ and allocated to processors. Throughout the implementation process, the results of each phase are verified. Maintenance techniques can be applied to manage the ADS application software. While we discuss the overall framework for the software development for ADS, our focus will be on the requirements analysis, verification, system design and the CASE tools supporting these phases.

2.2.4 Object-Oriented Requirements Analysis

To generate a precise, concise, understandable and correct model of a real-world application problem, we must examine its requirements, analyze their implications and restate them rigorously [6]. The analysis model consists of the object and dynamic models.

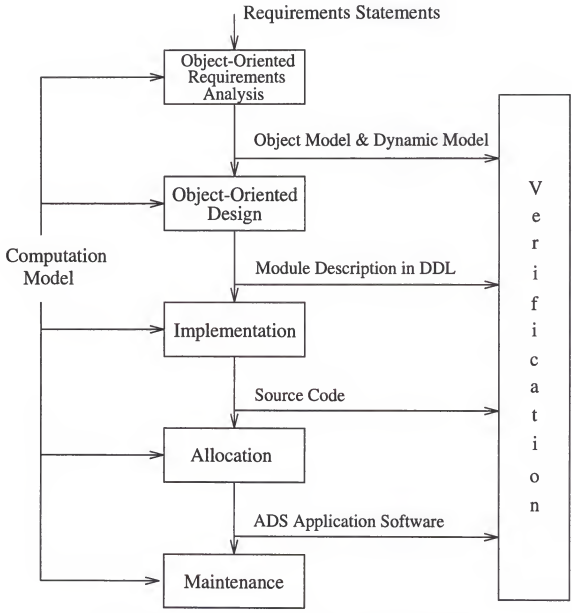

Figure 2.4. Software development framework for ADSs.

The object model can be constructed as follows [6]:

1. identify object classes from the requirement statements.
2. identify associations between classes.
3. identify object attributes.
4. organize classes using inheritance to share common structure.

We iterate the above steps until the object model is correct. Finally, we apply top-down and/or bottom-up approaches to build our hierarchical object model.

In real-world applications, software systems have a substantial number of object classes and associations. To accommodate this fact, we developed a hierarchy to "layer" the object model as follows:

After identifying object classes and associations, we group the classes using the following bottom-up guidelines:

- Tightly-coupled classes which frequently communicate with each other are grouped together.
- Each association should generally be shown on a single screen, but some classes must be shown more than once to connect different sheets.
- An attempt should be made to minimize the number of *bridge* classes, i.e., those classes which form the bridge between two screens or subjects (which are logical constructs for grouping classes). An efficient way to accomplish this is to look for *cut points* among the classes. A *cut point* is a class that is the sole connection between two disconnected paths of the object diagram. Such a class forms a better *bridge* between two sheets or subjects.

- A *star* pattern is frequently useful for organizing subjects. A single core subject contains the top-level structure of high-level classes. Other subjects expand each high-level class into a generalization hierarchy and add associations to additional low-level classes.

Because it is not realistic to apply an existing OORA approach directly to large and complex systems due to the complexity of the problem statement itself, we use the following top-down approach:

1. Identify the core activity areas of the system which require analysis, and provide a basis for work partitioning and paralleling development. Domain knowledge plays an important role in identifying these areas.

2. Divide the problem statements according to the core areas identified in the above step.

3. Apply our OORA approach to each of the core activity areas to generate the object and dynamic models for each area.

In most existing OORA methodologies, the results of the requirements analysis only concern static information such as class definitions, hierarchies, and relationships among classes. However, for distributed computing systems like ADS, more information, such as communication behavior, persistency of objects and role of objects, are needed in the requirements analysis phase and must be made available to the design phase. The dynamic model in [8] using a state transition diagram (STD) is not suitable for representing communication and checking consistency. We have developed a new dynamic model to enhance the original STD. It can be constructed as follows [6]:

1. prepare scenarios.

   We prepare the scenarios for typical interaction sequences. These scenarios show the major interactions, external display formats and information exchanges. The approach to constructing the dynamic model by scenarios ensures that important steps are not overlooked and that the overall flow of the interaction is smooth and correct.

2. construct event trace for each scenario.

   An event trace is an ordered list of events between objects. The scenarios are examined to identify all external events including all signals, inputs, decisions, interrupts, transitions and actions.

3. build an STD based on event trace.

   This starts with the event trace diagrams that affect the class being modeled. Every scenario or event trace corresponds to a path through the STD. Each branch in the control flow is represented by a state with more than one exit transition.

4. Verify consistency.

   Check for the completeness and consistency at the system level. Every event should have a sender and a receiver. Make sure that corresponding events on different STDs are consistent.

2.2.5 System Design

During the system design, we determine the structure of the system, which provides the context for more detailed decisions made in later design stages. By making high-level decisions for the entire system, the system designer partitions the problem into modules so that further work can be done by several designers

working independently on different modules [50]. In our computation model, ADS software is represented as a set of modules, and each module has its own control thread, a number of local objects and interfaces including import interface and export interface. Based on this structure, we derive the following procedure for the system design:

1. Identify all objects and all inter-object communication from the requirement statements, object model and dynamic model. At the requirements analysis phase, we focused on classes rather than objects. However, individual objects which play different roles in the system and their communication behavior should be identified at the system design phase.

2. Cluster the objects into modules. Based on our computation model, a module is not an object nor a function, but a package of objects, associations, operations, events and constraints that are interrelated and include a reasonably well-defined interface with other modules. The formation of the modules or the clustering of objects and related items directly influences the performance of the system being developed. A good bottom-up clustering algorithm will establish a modular design which will reduce inter-module communication cost, exploit potential concurrency among objects, and achieve fault-tolerance and reliability. For this purpose we use a clustering algorithm to systematically group the objects into modules according to a given set of criteria: minimizing communication, exploiting concurrency, functionality and user constraints.

3. Identify the control thread of each module. A control thread is a path through a set of state diagrams in which only a single object at a time is active. In our computation model, each module has only one control thread constructing

its own body, in which objects are initiated and the process activities of the module are defined.

4. Describe each module using DDL [4] defined in our computation model. Our DDL is used to represent software design for ADSs. The DDL representation of the software design reduces the gap between system design and implementation.

5. Decide boundary conditions. The boundary conditions, such as initialization, termination and failure, must be taken care of during system design. Constant data, parameters, global variables, tasks and others need to be initialized. In ADS software, content code [48] should be set for each module. On termination, internal objects can simply be abandoned, and each task must release any external resources it had reserved. In case of failure, a plan must be developed for orderly failure, as well as graceful exit on fatal bugs by leaving the remaining environment as clean as possible and recording or printing as much information about the failure as possible before terminating. Exception handling techniques for behaviors like retry, ignore, wait and time out are being developed in accordance with our computation model.

2.2.6 Implementation and Allocation

In our framework, coding is done using C++ programming language [51, 52, 53] with ADS constructs, such as invocation, receive_result, guard statement, etc., which are based on our computation model. The advantages of using ADS constructs are that the programmer will not have to be bothered by the synchronization, communication and/or location of processors. The ADS constructs can be translated automatically into C++ by a control mechanism in our computation model.

After the implementation, the modules must be allocated to processors in the ADS system. For the module allocation, we use the criteria of minimizing communication and balancing the load among the subsystems that constitute the distributed system [5]. Our module allocation algorithm is heuristic and suitable for allocating modules onto a distributed system whose subsystems are connected in the form of a Local Area Network (LAN) and communicating by means of broadcasting such as ADS. In this algorithm we try to achieve load balancing and minimize the communication thereby attempting to increase the concurrency. The algorithm includes fault tolerance aspects such as duplication, that is, no two duplicated modules are allocated to the same subsystem since this would defeat the purpose of duplication. Our approach also allows the user to specify constraints in allocation such as the allocation of a particular module to a specific subsystem.

2.2.7 CASE Environment

We have developed a CASE environment to aid the system analysts in generating the object and dynamic models from the given software requirements according to our OORA approach, and to aid the system developer in designing the system architecture according to our system design approach. This CASE environment includes the following components: a generic drawing editor using Graphical User Interface (GUI), two levels of integration, such as common user interface and transferability of data among all the tools in the environment, constraint maintenance which checks the connectivity between components, error checking, and automation of output generation from the CASE tools for OORA and for system design

Our goal is to provide a CASE environment for ADS application software development as shown in Figure 2.5, where two levels of integration are supported. The first level is the common user interface. Commands and tools are accessed uniformly

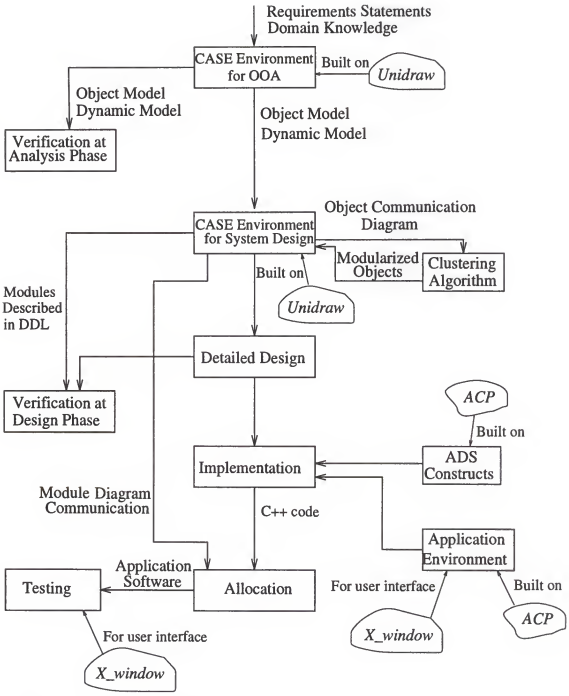

Figure 2.5. A CASE environment for ADS software development framework.

in both the tools for OORA, such as the object modeling tool, dynamic modeling tool and event trace editor, and for system design, such as the object communication modeling tool and module modeling tool. The next level of integration is the transferability of data between the tools, which is another kind of bridge. The CASE tools need to communicate with each other in order to retrieve information and check consistency. For example, the CASE tools for the system design may need information from the CASE tools for OORA, such as class definitions and communication behavior.

The design of our CASE tool is based on the framework of Unidraw [54]. These tools are running on X windows [55] and written in C++ [51, 52, 53]. We also used some objects from the object-oriented graphical toolkit InterViews [56], such as dialog boxes, text editor and structured graphics.

The object modeling tool consists of a viewer, a pull-down menu containing controls for executing specific commands, controls for engaging the current tool, a panner for panning and zooming the viewer and state variable views that display the values of state variables maintained by the object modeling tool. A state variable includes the name of the object model and modification status. The user engages the current tool by clicking on the appropriate control on the tool's left edge. The dynamic modeling tool is very similar to the object modeling tool. It also consists of state variable views, pull-down menu, viewer and panner. We can build the components of the dynamic model, such as state, event, class and transition, and manipulate components, such as move and select using this tool. In the implementation phase, our computation model provides language constructs for ADS software that control communication, synchronization and other ADS properties. These ADS constructs are translated into C++ by a preprocessor. The preprocessor scans the ADS application modules, checks the syntax according to

the syntax defined by the computation model and generates source code. The allocation algorithm has been implemented under a X window environment. The algorithm consists of three main procedures: *Cluster*, *Spillover* and *Update_Load*. The procedure *Cluster* clusters two nodes to form a compound node. This is done by merging the two nodes, a pivot and a non-pivot node. The procedure *Spillover* is for achieving load balancing. It checks whether a module can be safely allocated to a subsystem. Each time an allocation is performed, the load on the subsystem is updated by a call to the procedure *Update_load*. Finally, we are going to construct CASE environment tools and an integrated comprehensive set of services to support software development for distributed computing systems.

2.3 Related Work

Many techniques have been developed to help analysts verify the requirements specification such as Petri nets [57], and formal methods including temporal logic [58, 59], Z [60], VDM [61], and others. Most of these techniques require that a proof be constructed in order to show that the requirements specification actually meets its requirements statements. The petri net is a powerful and rigorous formalism to describe the requirements specification for distributed computing systems, and has received growing attention due to its graphical expressiveness and the possibilities of carrying out the general verification. Berthmomieu and Diaz [62] developed a method which allowed one to formally verify a time dependent system using Timed Petri Nets. Their method is related to the reachability analysis method. However, the petri net has limitations such that 1) it could be used to simulate specifications, but not to prove their properties, and 2) it is difficult to manage the complexity of the system. In formal methods [58, 60, 61], the proof is constructed using various axioms and inferences rules such as temporal logic to show that a requirements

specification meets its requirements statements. Unlike natural languages, each formal method has a precise syntax and semantics. Each formal method achieves 1) completeness by eliminating arbitrary assumptions or undefined concepts, 2) consistency by emphasizing contradictions present in the requirements specification, and 3) unambiguity by avoiding unclear function definitions. However, it also has limitations in the requirements verification. The requirements specification in a formal method is too rigorous and lacks a user interface. Because of these disadvantages, it is not appropriate to compare requirements statements written in a natural language with the formal requirements specification. Gerber and Lee [63] and Fidge [64] showed approaches which verified real-time properties from the requirements specification. However, their approaches presented verification for real-time systems, not for distributed computing systems. In addition, these approaches are not suitable for supporting verification for distributed computing systems.

Although much research in requirements verification has been done [58, 62, 63, 64], the verification of OORS for distributed computing systems has not been studied. In order to realize the full potential of object-oriented software development, we need to develop an effective verification approach for OORS.

2.4 Contributions of the Dissertation

In developing a verification approach to object-oriented requirements specification in software development for distributed computing systems, this dissertation makes the following contributions:

- I can contribute to the development of reliable software for distributed computing systems. Considering that many errors in software stem from errors in the requirements statements, this approach can reduce software development

cost by identifying potential problems at the requirements analysis phase of the software development cycle.

- In our formal object-oriented requirements specification language, the object-oriented concepts have been integrated to the formal state transition specification without sacrificing the advantages of either. Additionally, our language offers the following advantages: 1) It supports an abstract data type and the inheritance mechanism. In order to fully support an object-orientation, it can specify the inheritance relationship among classes. It allows single and multiple inheritances. 2) The attributes of a class are generic with their given types as parameters. 3) The behaviors of the system are represented by the state transition specification axiomatically. The state transition specification describes the change of states, the input that triggers the transition and the output with which the system responds. Each transition may have a *guard* which is a predicate.

- User-friendly graphical notations, i.e., inheritance graph, information tree and transition trace table, have been developed. Utilizing these graphical notations will facilitate the use of formal methods in real-world software development projects. The inheritance graph has a tree structure which contains only classes and represents the inheritance relationship among classes. The information tree is a tree which describes the static characteristics of classes and relationships among classes. It represents the classes, associations and aggregations among classes, as well as the methods and attributes of each class. The transition trace table describes an ordered list of transitions between different objects assigned to columns in a table. By scanning a particular column in the trace, we can see the transitions that directly affect a particular object.

Transformation procedures which ensure that there is an exact translation from formal specification to graphical notations have been developed.

- An approach to verify the consistency and completeness between requirements statements written in natural language and object-oriented requirements specification has been developed. It is a two-way approach: top-down and bottom-up. The top-down approach proceeds from the original requirements statements and checks the consistency of each requirements statement with the information in the graphical notations. In the bottom-up approach, the consistency of information in the graphical notations is checked by comparing it with the original requirements statements. We repeat the above steps for all information to verify the completeness of the OORS.

CHAPTER 3
OVERVIEW OF OUR APPROACH

In this approach, the verification of requirements specification is done by checking the completeness and consistency between the requirements statements in a natural language and the Object-Oriented Requirements Specification (OORS), which is expressed in terms of the object model [8] and dynamic model [8] generated by Object-Oriented Analysis (OOA). To do so, the OORS is transformed into a formal specification using a formal specification language, which is then transformed to graphic notations called the inheritance graph, information tree and transition trace table. Then the completeness and consistency of the requirements specification is verified by comparing the information in the graphical notations with the given requirements statements in the natural language. Our overall approach can be depicted as shown in Figure 3.1.

The input of our approach is 1) OORS expressed in terms of the object model and dynamic model, and 2) the requirements statements in the natural language given by the client. The object model is the static model which specifies the objects, classes, and their various relationships. It is represented by the object diagrams with classes and their structure, both being derived from the given requirements statements and domain knowledge. It can be generated by identifying object classes, associations between classes, attributes and inheritance. The dynamic model specifies class states, state transitions, class behavior and objects interaction. The dynamic model is represented graphically with the state diagrams. Each state diagram shows the state and event sequences in a system for one class of objects. Each model describes one aspect of the system, but contains references to the other models. The

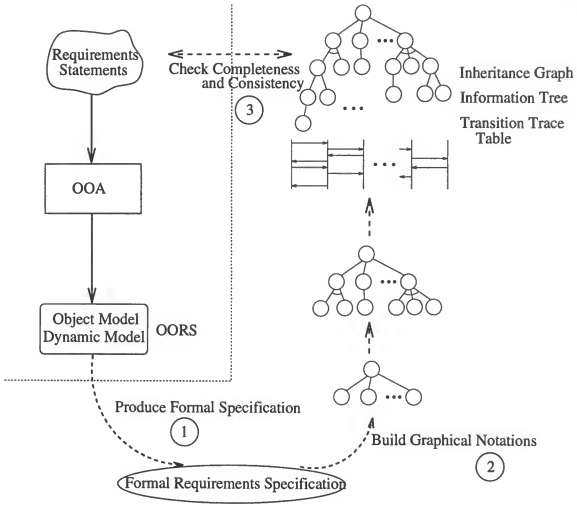

Figure 3.1. Our verification approach.

object model describes the data structure that the dynamic model operates on. The operations in the object model correspond to events in the dynamic model. The dynamic model describes the control structure of objects. It shows which decisions depend on object values and which actions change object values.

The goal of this approach is to identify the inconsistency and/or incompleteness, if any, between the OORS and the requirements statements. The requirements statements are a high-level description of a software system in a natural language. The OORS describes a software system's external behavior – what inputs are expected by the software, what outputs will be generated by the software, and the details of relationships that exist between those inputs and outputs. During the OOA, we expand the OORS using the domain knowledge and ignore redundant or unnecessary information from the given requirements statements. Some differences may exist between requirements statements and OORS because they are at different levels of detail in specifying the requirements of the system. Thus, such differences should be identified and their effects on the system are evaluated during the verification of OORS. Our verification approach will identify the missing information in the OORS which may be deleted as unnecessary information and the information in the OORS which is not specified in the requirements statements. Such information may be created from the domain knowledge or as a result of mistakes or features introduced during the OOA.

Our verification approach can be summarized as follows:

Step 1 Produce the formal requirements specification described by a formal specification language from the derived OORS.

Step 2 Build graphical notations from the formal specification obtained in **Step 1**.

Step 3 Apply the top-down and bottom-up approaches to check the completeness and consistency between requirements statements and OORS.

The use of formal specification methods in software development enhances the insight into and understanding of software requirements, and helps clarify the customer's requirements by revealing or avoiding contradictions and ambiguities in the specifications. It also enables rigorous verification of specifications and their software implementations, and facilitates the transition from specification and design to implementation. Thus the use of formal methods is expected to lead to increased software quality and reliability. Moreover, early verification of specifications would increase specification quality thereby reducing life cycle costs.

Our formal object-oriented requirements specification consists of following portions:

- class name
- inheritance classes
- attributes, associations, aggregations and constraints
- methods
- dynamic behavior, i.e., state transitions

Our formal object-oriented specification language supports an abstract data type and the inheritance mechanism in order to fully support an object-orientation. The attributes of a class are generic with their given types as parameters. That means that a class can serve as a template for other classes, in which the template may be parameterized by other classes, objects, and/or methods. A generic class must be instantiated (its parameters filled in) before objects are created. The specification

of input and output parameters for each method in a class includes class variables as well as local variables. The appropriate output may not be just a function of the input, it may also be a function of the current state of the class. The dynamic behavior of an object is described by state transitions. This naturally follows from the fact that every state transition is triggered by an event and in turn changes the state of at least one instance.

However, the notation and the conceptual grammar of formal specification languages require familiarity with discrete mathematics and symbolic logic which most practicing software engineers, designers, and implementors do not currently have. Most software engineers have not been trained in techniques required to develop formal software specifications, and their inexperience in these techniques makes formal specification development difficult. Non-formal methods exist for the development of object-oriented systems [65]. Attempts to enhance these with mathematical semantics and reasoning techniques have been made [66], which could potentially lead to formal specification techniques supported by diagrammatic notations. I believe that such an integration of formal and informal notations is an essential support for development and validation of specifications, particularly at the domain and system analysis stage.

In this approach to be presented here, there are three different graphical representations derived from formal specification: the inheritance graph, the information tree and the transition trace table. The inheritance graph contains only classes and represents the inheritance relationships among classes. It is not a tree, but a graph to support multiple inheritance. Information tree describes the static characteristics of classes and relationships among classes. It represents the classes, associations between classes, their methods and attributes. The transition trace table describes the dynamic behavior of each class. It is an ordered list of events between different

objects assigned to columns in a table. I assign a separate column to each object. By scanning a particular column in the trace, we can see the events that directly affect a particular object.

Finally, I check the completeness and consistency between requirements specification which is depicted in the graphical notations, such as inheritance graph, information tree and transition trace table, and requirements statements given by clients. I apply two different approaches: top-down and bottom-up. In the top-down approach, I identify each statement in the requirements statements and the corresponding information from the graphical notations, and then compare two different representations. In the bottom-up approach, I identify each item of information in the graphical notations derived from the OORS and compare it with a statement in the requirements statements. That is, for each statement in the requirements statements, I check the consistency by comparing it with information in the graphical representations and I also check the consistency by looking for the statement in the requirements statements which has the same meaning for each piece of information in the graphical representations. I verify the completeness by checking all statements in the requirements statements and all information from the OORS.

In the following chapters, I will discuss each step of our requirements verification approach in more detail.

CHAPTER 4
FORMAL REQUIREMENTS SPECIFICATION

Formal requirements specification is a specification expressed in a language whose vocabulary, syntax, and semantics are formally defined, and which has a mathematical basis, usually in formal logic [67]. As previously mentioned, the use of formal specification methods in software development enhances the insight into and understanding of software requirements, and helps clarify the customer's requirements by revealing or avoiding contradictions and ambiguities in the specifications. It also enables rigorous verification of specifications and their software implementations, and facilitates the transition from specification and design to implementation. Thus the use of formal methods is expected to lead to increased software quality and reliability. Moreover, early verification of specifications would increase specification quality thereby reducing life cycle costs.

As formal specifications can be analyzed using mathematical operators, mathematical proof procedures can be used to test and prove internal consistency and syntactic correctness of specifications. Furthermore, the completeness of the specifications can be checked in the sense that all enumerated options and elements have been specified. The formal proof procedures can also be used to verify if a design or its implementation satisfies its antecedent specifications.

The concern of the object-oriented development approach is to create generalized, adaptable and re-usable systems. This is consistent with the aim of formal specification to create abstract functional specifications of a system, in a form which enables formal verification and which supports validation. The object-oriented specification languages allow explicit encapsulation of state and associated methods into

a class, and hence extend the amount of specification information which can be formally represented, rather than informally in textual annotations. In addition, they support compartmentalization of proofs, since only properties of classes used directly or indirectly in a given class will be relevant for the proving of properties about that class. Some aspects of object orientation, such as message passing as the paradigm for communication between objects, will not be directly relevant to a specification language, since one of the purposes of a specification language is to abstract away from possible implementations in order to gain generality and re-usability, in addition to clarity of expression.

In this chapter, I will explain the ambiguity problems in the requirements statements, our formal object-oriented requirements specification language and procedure for producing formal specification from the OORS.

4.1 Ambiguities in the Requirements Statements

4.1.1 Definition of Ambiguity

A software requirements specification is unambiguous if and only if every requirement stated therein has only one interpretation [68]. As a minimum, this requires that each characteristic of the final product be described using a single unique term. Or, in cases where a term used in a particular context could have multiple meanings, the term must be included in a glossary where its meaning is made more specific.

Natural Language Pitfalls

Requirements are often written in a natural language (for example, English). Software requirements specification writers who use a natural language must be especially careful to review their requirements for ambiguity [69]. The followings are examples:

1. The specification *The data set will contain an end of file character*, might read as

   (a) There will be one and only one end of file character.

   (b) Some character will be designated as an end of file character.

   (c) There will be at least one end of file character.

2. The specification *The control total is taken from the last record*, might be read as

   (a) The control total is taken from the record at the end of the file.

   (b) The control total is taken from the latest record.

   (c) The control total is taken from the previous record.

3. The specification *All customers have the same control field*, might be read as

   (a) All customers have the same value in their control field.

   (b) All customer control fields have the same format.

   (c) One control field is issued for all customers.

4. The specification *All files are controlled by a file control block*, might read as

   (a) One control block controls the entire set of files.

   (b) Each file has its own block.

   (c) Each file is controlled by a control block, but one control block might control more than one file.

4.1.2 Sources of Ambiguity in Requirements

Whenever we use tools that ignore the human aspect, we describe requirements imperfectly and create ambiguities. Ambiguities, in turn, lead to diverse interpretations of the same requirement. I will first discuss some of the ambiguity encountered in the requirements.

The source of ambiguities in requirements statements are missing requirements, ambiguous words and introduced elements [70]. For instance, igloo, castle and space station are three rather different structures built in response to the same ambiguous requirements statements:

Create a means for protecting a small group of human beings from the hostile elements of their environment.

Examining the differences among the solutions, we find clues to some of the ambiguity in the requirements.

1. Sometimes requirements are missing. For instance, there is no requirement concerning properties of materials, such as local availability, durability, or cost. Thus, it's not surprising the three solutions differ widely in their use of materials. The requirements statement is equally ambiguous as to the *structure*, or how the building materials will be assembled. We don't know the desired size, shape, weight, or longevity of the structure. Little is said or even implied about what *functions* will be performed inside these structures, leaving open the question of specific functional elements, such as stoves, servant's quarters, beds, and ballrooms. Nothing is said about the *physical environment*, either internal or external. The structure could reside on land, sea, or in the air, on an ice park, or even in outer space. Then, too, we know nothing about the specific hostilities from which we are to protect the inhabitants. What

about the social and cultural environment? Is this small group of human beings a family unit, and if so, just what constitutes a family unit in this particular culture? Perhaps it is a working group, such as hunters or petroleum engineers, or possibly a recreational group, such as poker players or square dancers.

2. Even when requirements are stated explicitly, they may use ambiguous words. For instance, the word "small" does not adequately specify the *size of the group*. Beware of comparative words, like "small" or "inexpensive", in requirements statements. A group of 25,000 would be "small" if we're talking about football fans at a University of Florida home game, where a new football stadium could fulfill the stated requirements. Another dangerous word in the requirements statement is "group", which implies that the people will interact, somehow, but it's not clear *how*. Designing a structure for one group would be quite different from designing for an other. Even the term "structure" carries a load of ambiguity. Some readers would infer that "structure" means something hard, durable, solid, opaque, and possibly heavy. If we unconsciously accept that inference, we subliminally conclude the problem is to be solved with traditional building materials, thus limiting the range of possible effective designs.

3. Of course, we can guard against ambiguous words by carefully exploring alternative meanings for each word in the requirements statement, but that won't protect us from another problem. The term "structure" never actually appeared in the requirements statement, but somehow slipped into our discussion without our noticing. The requirements statement actually says "create a means", not "design a structure". Some requirements ambiguities are so

obvious that they would be resolved in the casual designer-client conversations long before the actual design process began. More subtle ambiguities, however, may be resolved unconsciously in the designer's mind. In this case, an innovative, but nonstructural, "means" of protecting a small group might be overlooked. For instance, the designer might

1) protect against rain by electrostatically charging the raindrops and repelling them with electrical fields.

2) protect against belligerent crowds by supplying aphorisms, such as "Sticks and stones may break my bones, but names will never hurt me", or "I may have to respond to what other people do, but they don't define me".

3) protect against winter by moving south, and against summer by moving north.

4.1.3 Cost of Ambiguity

Billions of dollars are squandered each year building products that don't meet requirements, mostly because the requirements were never clearly understood. For instance, Boehm [71] analyzed sixty-three software development projects by various firms, including IBM, GTE, and TRW and determined the ranges in cost for errors created by false assumptions in the requirements phase but not detected until later phases. (See Table 4.1.)

Although Table 4.1 vividly shows the importance of detecting ambiguities in requirements. First of all, Boehm studied only projects that were completed, but some observers have estimated that approximately one-third of large software projects are never completed. Much of the enormous loss from these aborted projects can be attributed to poor requirements definition.

Table 4.1. Relative cost to fix an error.

Phase in which found	Cost ratio
Requirements	1
Design	3-6
Coding	10
Development testing	15-40
Acceptance testing	30-70
Operation	40-1000

4.1.4 Existing Methods Solving Ambiguities in Requirements

Formal Requirements Specifications Languages

One way to avoid the ambiguity inherent in natural language is to write the requirements specification in a formal requirements specification language. Formal specification techniques are generally beneficial because a formal language makes specifications more concise and explicit. These techniques help us acquire greater insights into the system design, dispel ambiguities, maintain abstraction levels, and determine both our approach to the problem as well as its implementation.

- One major advantage in the use of such languages is the reduction of ambiguity. This occurs, in part, because the formal language processors automatically detect many lexical, syntactic, and semantic errors.
- One major disadvantage in the use of such languages is the length of time required to learn them.

Rapid Prototyping

Prototyping is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem [68]. It is a partial implementation because if it were a full

implementation, it would be the system, not a prototype of it. The purpose of the partial implementation is a fundamental part of the definition and differentiates a prototype from a subset. A subset is also a partial implementation; however, the purpose of a *subset* is to provide early functionality, whereas the purpose of the prototype is to learn something. Thus a prototype could be a subset, and vice versa, but they are not the same.

There are two kinds of prototyping – rapid and evolutionary. In the *rapid* approach [72, 73, 74], the prototype software is constructed in order to learn more about the problem or its solution and is usually discarded after the desired knowledge is gained. In the *evolutionary* approach [75, 76, 77, 78], the prototype is also constructed in order to learn more about the problem or its solution, but once the prototype has been used and the requisite knowledge gained, the prototype is adapted to reflect the better understanding of the clients' needs. The cycle is then repeated indefinitely until the prototype system satisfies all needs and has thus evolved into the real system.

A rapid prototype can be built during any synthesis phase in the software life cycle. During the requirements phase, a quick-and-dirty rapid prototype can be constructed and given to a potential user or customer in order to (1) determine the feasibility of a requirement, (2) validate that a particular function is really necessary, (3) uncover missing requirements, or (4) determine the viability of a user interface. It is one way to avoid the ambiguity inherent in natural language. The prototype should be *quick-and-dirty*. Its development should be quick because its advantage exists only if results from its use are available in a timely fashion. It should be dirty because there is no justification for building quality into a product that will be discarded. The common scenario for a rapid requirements prototype calls for (1) writing a preliminary SRS, (2) implementing the prototype based on

those requirements, (3) achieving user experience with the prototype, (4) writing the real SRS and then, (5) developing the real product.

4.2 Formal Object-Oriented Requirements Specification Language

Object-oriented software development and formal methods support each other [79]. The complexity of the software that object-oriented methods are built for requires rigorous methods to meet the stringent quality criteria. Formal methods offer this kind of rigor. Conversely, formal software development methods may only be successful when they include appropriate support for traditional software engineering principles, as modularity and reuse. Object-orientation is the most advanced approach in this direction. Our formal object-oriented requirements specification language is a specification language which combines formal methods and object-orientation.

A formal object-oriented specification language used to describe OORS must have the following characteristics:

- It must support an abstract data type. That is, a class represents an abstract data type.

- It must support the inheritance mechanism. In order to fully support an object-orientation, it can specify the inheritance relationship between the classes. It allows single and multiple inheritances. Multiple inheritance permits a class to have more than one superclass and to inherit features from all the parents. This permits mixing of information from two or more sources. The advantages of multiple inheritance are greater power in specifying classes and an increased opportunity for reuse. For instance, if we write

  class spec C **is** ... **inherits** D ...

we mean the operation "add all features of D to the features of C". These features comprise attributes and methods.

- The attributes of a class must be generic with their given types as parameters. That means that a class can serve as a template for other classes, in which the template may be parameterized by other classes, objects, and/or methods. A generic class must be instantiated (its parameters filled in) before objects are created. The attribute consists of constants, variables and their types. Attributes also are represented with constraints which restrict its value or the range.

- The specification of input and output parameters for each method includes class variables as well as local variables.

  $m : S \times I \longmapsto S' \times O$

  S, S' : class variable (representing class state)

  I : local variables (for input domain)

  O : local variables (for output domain)

  The appropriate output may not be just a function of the input; it may also be a function of the current state of the class. In other words, the method (m) must establish a complete mapping from the cross product of the input domain (I) and state domain (S) into the cross product of the output domain (O) and the state domain (S') (that is, $S \times I \longmapsto S' \times O$).

The formal specification language I use for describing OORS is an extension of that by Breu [80]. In order to support parameterized classes, I extend the **attributes** portion [80] with genericity features. The specification of input and output

parameters for each method includes a class variable to represent the class state. It also combines state transition specifications and object-oriented specifications, such as classes.

The OORS $Spec[Sys]$ of a software system Sys can be described by the specification $Spec[C_i]$ of classes $C_i, i = 1, 2, \cdots, n$. Let a software system Sys consist of classes $C_i, i = 1, 2, \cdots, n$. We have

$$\begin{aligned} Spec[Sys] &= Spec[C_1, C_2, \ldots, C_n] \\ &= Spec[C_1] + Spec[C_2] + \ldots + Spec[C_n] \end{aligned}$$

A class specification $Spec[C_i]$ is defined as follows:

class spec C_i **is**

inherits $I_1, I_2, \ldots, I_l$

attributes $A_1, A_2, \ldots, A_m$

methods $M_1, M_2, \ldots, M_p$

axioms $T_1, T_2, \ldots, T_q$

end class spec

where

$C_i, 1 \leq i \leq n$ represents the name of a class.

$I_i, 1 \leq i \leq l$ represents the name of a superclass.

$A_i, 1 \leq i \leq m$ represents the attribute variable and its type, and the relationship variable which describes an association or aggregation between the classes.

$M_i, 1 \leq i \leq p$ represents the name of a method, its type of parameters and operation of its method. **create** method is also involved.

$T_i, 1 \leq i \leq q$ represents the state transitions based on the dynamic behavior showing the events the object receives and sends.

Our formal specification language uses an state transition specification to describe the behavior of the system. The **axioms** portion in the class specification represent the state transition specification axiomatically. The state transition specification describes the change of states, including the input that triggers the transition and the output with which the system responds. In response to an input, the system generates an output and changes state. Both the output (O) and the next state (S_N) are purely functions of the current state (S_C) and the input (I). Thus

$$S_N = F(S_C, I)$$
$$O = G(S_C, I).$$

Each transition may have a *guard* which is a predicate. When the guard is true, the transition is said to be *enabled* for them. When the guard is false, its operation is delayed until it is true. The guard is given after the keyword **when**.

4.3 Producing Formal Specification from OOA

In this section I will give the way that analysis models formulated in OOA technique can be mapped into formal object-oriented specifications. Our formal object-oriented specification language which is illustrated in the above section will be used as the formal language.

4.3.1 Handling Ambiguities

I have studied the methodology to check the testability of requirements specification (that is, object and dynamic models) during the transformation of requirements specification into formal requirements specification to ensure that there are no ambiguities in the requirements specification. The requirements specification are

testable if and only if there exists some finite cost-effective process with which a person or machine can check that the requirements specification meet the requirements [69]. Testable requirements specification are those which are specific and unambiguous, with a clearly identifiable result when they are met. For instance, we specify that accuracy should be "within $\pm$ 1" rather than "sufficient to meet mission requirements" in a testable requirements specification.

Our formal requirements specification language consists of following portions:

- class name
- inheritance classes
- attributes and constraints
- relationships, such as associations and aggregations
- methods
- dynamic behavior

When I transform the requirements specification to formal specification, I check the testability of each piece of information in the requirements specification to ensure that there are no ambiguities in the requirements specification. The class name and inheritance class portions of our formal language are obvious because all objects have an identity and are distinguishable. The attribute and constraint portions have a high possibility of ambiguities because they represent the characteristics desired by the client and are thought of as adjectives or adverbs in the requirements statements. When I transform the values of attributes of a class to the attribute variables of formal specification, I check whether the information is testable or not. That is, I check that the information is represented by measurable quantities,

not unclear words. The relationships between classes may also have ambiguities because the relationships, such as associations and aggregations between classes, represent the structures of a system and are thought of as verbs or verb phrases in the requirements statements. When I transform the relationships to the association or aggregation variables at the **attributes** portion of formal specification, I check that the relationship is clear, that is, whether it has multiple interpretations or not. Because the method portions represent the method name and its domain and range types, it does not have any ambiguities. The dynamic behavior portions have ambiguities because these portions can be generated with user's desire, domain knowledge and designer's mind. Every state transition changes the state of at least one instance and is triggered by an event. When I transform a transition in the dynamic model into a transition based formal language, I check the source of state alternation as identifying the input event that causes the transition and the object which requests the input event.

4.3.2 Formalizing Object Model Notation

Each component of OOA object model notation will be considered in turn, together with the choices that exist for their formalization.

Objects and Classes

All objects in an object model are identified and are distinguished whether it is active or passive. The active object is an object which can initiate a process by invoking a method or methods. The passive object is an object which has methods which must be invoked by the request of another object(s).

Attributes

An attribute is a data value held by the objects in a class [8]. *Name*, *age*, and *weight* are attributes of *Person* objects. Each attribute has a value for each object instance. For example, the attribute *age* has the value "33" in the object *Keunhyuk Yeom*. Different object instances may have the same or different values for a given attribute. Each attribute name is unique within a class. The attributes of a class C in an object model become attributes of the formal object class C expressing this class. I define the appropriate types for attributes. Domains which we do not wish to detail further can be defined as *generic type*(s) in our formal object-oriented specification language and refined later as needed.

Associations

An association describes a group of physical or conceptual connections between object instances with common structure and common semantics. That is, it is a means for establishing a relationship between object classes [8]. For example, a person *Works-for* a company. All the connections in an association connect objects from the same classes. Associations are inherently bidirectional. If we have a pair C, D of classes, with an association r between C and D, then we form formal specification classes for each class, C and D, involved in the association, and a class R encapsulating a system involving a set of instances of C and a set of instances of D, and an attribute which is a relationship between these sets.

class spec C **is**

attributes

$r : R$

attributes of C

...

end class spec

class spec D **is**

attributes

$r : R$

attributes of D

...

end class spec

class spec R **is**

attributes

$r : C, D$

attributes of R

...

end class spec

The use of generic classes could be made to express general forms of association. For example, C and D in the above class could be replaced by the generic class parameters X and Y of R.

As an example of our approach, consider a domain where I am defining entity types *House*, *HouseOwner* and an attribute association *Ownership* between them (Figure 4.1).

This situation can be formally expressed by the classes:

class spec *HomeOwner* **is**

attributes

ownership : *Ownership*

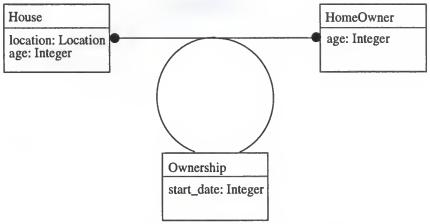

Figure 4.1. Home-ownership representation.

age : Integer

...

end class spec

class spec *Ownership* **is**

attributes

ownership : *HomeOwner*, *House*

start_date : Integer

...

end class spec

class spec *House* **is**

attributes

ownership : *Ownership*

location : Location

age : Integer

...

end class spec

Aggregations

Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. One common example is the bill-of-materials or parts explosion tree. Aggregations are modeled via class composition in our formal object-oriented specification language. If we have a class C in object model which has aggregations r and s to classes D and E, respectively, then we simply define:

class spec C **is**
  attributes
    $r : D$
    $s : E$
    other attributes of C
  ...
end class spec

Constraints

Constraints are functional relationships between entities of an object model which restrict the values those entities can assume. The term *entity* includes objects, classes, attributes, and associations. Constraints are modeled by predicates limiting the formal items corresponding to the object model items upon which the constraint is placed. If there are constraints on the relation between two associations, then these associations need to be represented as attributes of a single class. The same is true of more complex examples. Constraints involving time can be translated into temporal [58, 59] constraints of a class. For instance, the statement that "if M

occurs, then N can never occur again" is directly expressed by the temporal logic formula

$$\Box(M \Rightarrow \Box(\neg N)).$$

Inheritance

Inheritance is a mechanism for incremental or relative specification, whereby new classes may be derived from one or more existing classes. Subject to certain constraints, the properties of inherited classes are also inherited. Inheritance therefore is particularly significant in the effective reuse of existing specification. Our formal object-oriented specification language supports *multiple inheritance* which permits a class to have more than one superclass and to inherit features from all the parents. Inheritance is modeled at the **inherits** portion in the formal requirements specification.

Operations and Methods

An operation is a function or transformation that may be applied to or by objects in a class. All objects in a class share the same operations. Each operation has a target object as an implicit argument. The behavior of the operation depends on the class of its target. I capture all functions and put them into the **methods** portion of the formal requirements specification.

4.3.3 Formalizing Dynamic Model

Given a formal requirements specification of an OOA object model, the next step is to formally specify how the state may change according to the OOA dynamic models (i.e., state transition models). The dynamic model consists of multiple state diagrams, with one state diagram for each class with important dynamic behavior, and shows the pattern of activity for an entire system. A state diagram relates

events and states. When an event is received, the next state depends both on the current state and the event. A change of state caused by an event is called a transition. The state diagram specifies the state sequence caused by an event sequence.

The syntax and semantics of our formal requirements specification language are explained in the above section. The **axioms** portions of the class specification represent the state transitions axiomatically. Our approach to modeling state transition in formal specification is simply to specify a schema for every state transition. The general state transition structure of our formal specification is given by the schema *Transition* below.

$$Transition(S \times I(obj) \longmapsto S' \times O_1(obj_1) \times O_2(obj_2) \times \cdots \times O_n(obj_n))$$

As every state transition changes the state of at least one instance, and is triggered by an event, the components of every state transition schema must include:

- The state (S) being altered by the transition.
- The input event (I) that causes the transition and the object (obj) which requests the input event.
- The state (S') after a transition.
- The output events (O_1, O_2, ... and O_n) that are generated by the action associated with the transition and destination objects (obj_1, obj_2, ... and obj_n).

A transition may occur by any one of a collection of events. To allow for this capability, a **select** statement is used. The **select** statement provides a transition with a means for selecting from a set of alternatives. The syntax is

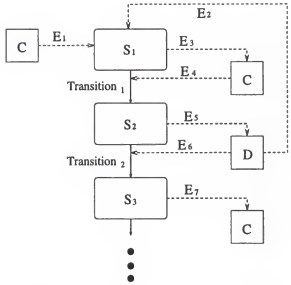

Figure 4.2. State transition diagram.

select

Transition ($S \times I_1 \longmapsto S' \times O$)

Transition ($S \times I_2 \longmapsto S' \times O$)

Transition ($S \times I_3 \longmapsto S' \times O$)

end select

where O is a set of output events

For instance, in Figure 4.2 the transition from a state S_1 is triggered by one of two input events, E_1 from class C or E_2 from class D. Thus, the state transition specification for Figure 4.2 can be described as follows:

select

Transition_1 ($S_1 \times E_1(C) \longmapsto S_2 \times E_3(C)$)

Transition_1 ($S_1 \times E_2(D) \longmapsto S_2 \times E_3(C)$)

end select

Transition_2 ($S_2 \times E_4(C) \longmapsto S_3 \times E_5(D)$)

The order of transitions describes all the sequences of methods in a system that are concerned with time. That is, all the sequences of events that occur during the execution of a system are represented by transitions axiomatically. For instance, the transition from state S_1 to state S_2 is performed before the transition from state S_2 to state S_3 in Figure 4.2. That is,

Transition_1 $(S_1 \times E_1(C) \longmapsto S_2 \times E_3(C))$
Transition_2 $(S_2 \times E_4(C) \longmapsto S_3 \times E_5(D))$

4.3.4 Control Structure in the State Transition Specification

As I explained before, our formal object-oriented specification language has control structures, such as **select**, **if-then-else**, and **while**, to control the sequences of state transitions.

Selection

Selection control structures allow the designer to specify that a choice is to be made among a certain number of possible alternative transitions [81, 82]. The **if-then-else** statement is an example of a selection control structure that specifies the executions of a transition according to a boolean expression. A **then** and an **else** alternative allows the designer to choose transitions between two control paths as a consequence of a test.

The **if-then-else** statements allow nesting. Sometimes the nesting of **if-then-else** statements raises an ambiguity problem. In the example

if A $>$ 0 **then if** A $<$ 100 **then** T_1 **else** T_2

It is not clear whether the **else** branch is part of the innermost conditional (**if** $A < 100 \cdots$) or the outmost conditional (**if** $A > 0 \cdots$). To eliminate ambiguity, I automatically match an **else** branch to the closest conditional without an **else**.

The **select** statement, multiple-choice selection constructs, specifies selection among two or more branches based on the value of a boolean expression. For example

select
  { **when** (A) } T_1
  { **when** (B) } T_2
  { **when** (C) } T_3
end select

Iteration

Most useful control structures involve repetition of a number of transitions. Our formal specification language supports condition-driven loop constructs, that is, the repetition structures in which new iterations are executed until the value of a boolean expression is changed.

while B
  T_1
  T_2
  ⋮
  T_n
end while

The condition B is evaluated before executing the body of loop. The loop is exited if B is false.

If the loop is nested within other loops, exiting an inner loop may also result in the exiting any number of enclosing loops.

CHAPTER 5
GRAPHICAL NOTATIONS

This chapter describes the graphic notations that are used to represent the information visually from the formal requirements specification discussed in the previous chapter.

Formal requirements specifications help to clarify what each part is supposed to do, when the design process leads to insights which cause the change of the specification. They also enable us to define requirements in a rigorous, unambiguous and consistent fashion, and to prove the correctness of the final software solution. However, in the past most effort in formal specification research has been concerned with the development of formal notation and inference rules. Relatively little effort has been devoted to the development of methodological and tool support. Unless viable strategies for incorporating formal methods in the software development process are developed, it may be difficult to attain the promise of formal methods in real-world software development projects [83].

I developed graphic notations, such as inheritance graph, information tree and transition trace table, to verify the OORS.

5.1 Inheritance Graph

Inheritance is an important mechanism in object-oriented software development by which one class can be defined as a specialization of another. Here I build the inheritance structures, looking at the **inherits** portion of each class specification.

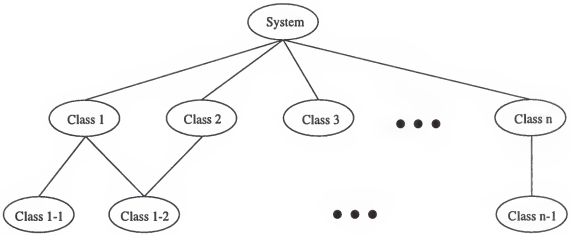

Figure 5.1. The structure of an inheritance graph.

5.1.1 Notation

The notation used for inheritance is a tree structure. Strictly speaking, it's not a tree, but a graph because our object-oriented software development methodology supports a multiple inheritance mechanism which permits a class to have more than one superclass and to inherit features from all the parents. This graph contains only classes and represents the inheritance relationship among them. A typical structure of an inheritance graph is shown in Figure 5.1. A root in the inheritance graph is a node which represents a system. Each node except the root node represents a class labeled with its own particular name. Subclasses are represented hierarchically. That is, the closer the node is to the root node, the higher it is on the graph. For example, a **Shape** class and two subclasses **Rectangle** and **Circle** are shown in Figure 5.2. The multiple inheritance mechanism is also shown in Figure 5.1, that is, *Class 1–2* is inherited from both *Class 1* and *Class 2* classes.

5.1.2 Development of the Inheritance Graph

The development of inheritance graph is relatively easy. The process for building it uses a bottom-up approach. That is, first I identify all classes from the formal

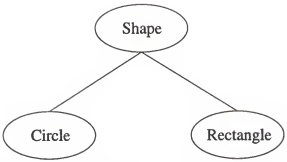

Figure 5.2. Shape and its subclasses in an inheritance graph.

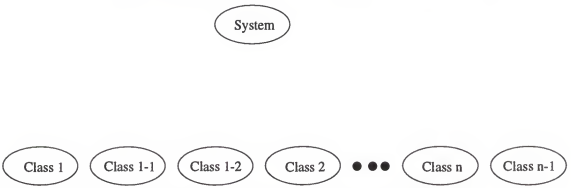

Figure 5.3. Inheritance graph after identifying classes.

requirements specification and make nodes for the classes as shown in Figure 5.3. Then, I construct an inheritance graph looking at the **inherits** portions in the formal requirements specification. For example, if class C has classes A and B at the **inherits** portion of the formal specification for class C, then class C becomes a child of class A and class B. Finally, each class node without any parents receives a direct connection to the system node.

5.2 Information Tree

An information tree is a tree which describes the static characteristics of classes and relationships between the classes. It represents the name of classes, associations and aggregations between classes, as well as the methods and attributes of each class.

5.2.1 Notation

I build an information tree from the formal requirements specification written in the formal specification language. I can explore all the information in the OORS by traversing the information tree. The root of the tree represents the system. The system consists of classes and each class has methods. Each method has attributes and constraints. The *create* method is involved in the method portions of an information tree. It is a method by which an object is instantiated from the class because a class describes a set of object instances of a given form. The *create* method also has attributes and their constraints. The attributes of the method, *create*, represent the properties of an object, that is, data values held by the objects in a class. A path in the information tree starts from the root node and traverses the tree through the class, method, attribute and constraint nodes. A relationship between classes represents a physical or conceptual connection between the classes. They are represented by dashed lines, that is, the invocation of a method in another class is represented by a dashed arrow line from the invoking class to the invoked class. The association relationships between the classes are also represented by dashed lines with arrows. A letter "D" between the method and the attribute nodes indicates that the attribute is a domain of the method. A letter "R" entails that the attribute is a range of the method. A typical structure of an information tree is shown in Figure 5.4.

5.2.2 Development of the Information Tree

The information tree is constructed from the **class** name, **attributes**, **methods** portions in the formal requirements specification. That is, the information tree describes the static structures of the OORS. The information tree can be built as follows.

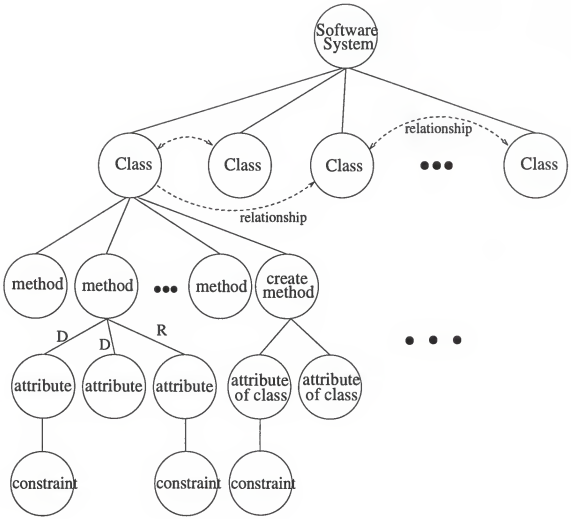

Figure 5.4. The structure of an information tree.

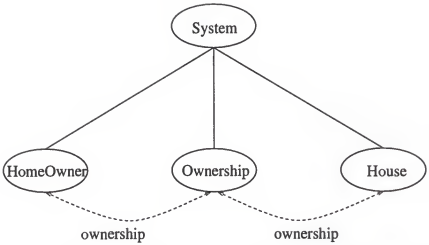

Figure 5.5. Information tree for home-ownership relationship.

1. Identify classes and their relationship from the formal requirements specification. Classes are identified from the class name in the formal requirements specification. I can identify the relationships between the classes from the method invocation of a class to other classes, and associations between them which appear at the **attributes** portion. For example, consider the Home-ownership representation in Figure 4.1. I can build an information tree representing association relationships among classes *House*, *HouseOwner* and *Ownership*. The class *HouseOwner* has an association relationship *ownership* to the class *Ownership* and the class *Ownership* has an association *ownership* to the class *House*. The information tree is shown in Figure 5.5.

2. Identify the methods (that is, functions) for each class from the **methods** portion of the formal requirements specification. Functions are the "what" of a product, describing what the product is to accomplish. I capture all functions, understand them, and make them children of the class in the information tree. The *create* method is also involved.

3. Identify the attributes of each method from the formal parameters of a method in the formal requirements specification. The attributes are characteristics which are required by the user or the client and the data values held by each object. They are represented as the attribute variables or as the formal parameters of a method. If the attribute is an input parameter of a method, I put a letter "D" on the edge between the method and the attribute. Similarly, if the attribute is an output parameter of a method, then I put a "R" between them. There may be a relationship between different attributes. Thus, different attributes may be interrelated and dependent. The attributes of each method are attributes of the entire software system and the same attribute may qualify more than one function. I also identify the attributes of a class from the **attributes** portion in the formal requirements specification and put them as the children of the *create* method node in an information tree. The attributes of the *create* method represent the properties of an object.

4. Identify the constraints of an attribute if they exist. I identify the constraints of attributes from the **attributes** portion in the formal requirements specification. The constraints of an attribute appear between the parentheses following an attribute variable in the formal requirements specification. For instance, when I create a window in the window system, I can give a restriction for ratio between width and length, that is, **length: integer ($0.8 \leq$ length/width ≤ 1.2)**. Constraints represent mandatory or boundary conditions placed on the attribute. The constraints of an attribute must be satisfied in software development in order to be real (that is, to be defined by the requirements). For instance, if the amount of withdraw in a day is restricted to $200 in the

Automated Teller Machine (ATM) system, the attribute *cash* of class *ATM* has constraints "less than or equal to $200".

5.3 Transition Trace Table

I can explore all the information in the OORS by traversing the information tree. However, the notations used in the information tree do not represent all dynamic behavior, most of which comes from the domain knowledge. I use a transition trace table to represent the dynamic behaviors of a system.

5.3.1 Notation

A transition trace is an ordered list of transitions between different objects assigned to columns in a table. This table shows each object as a vertical line and each transition as a horizontal arrow from the sender object to the receiver object. Time increases from top to bottom, but the spacing is irrelevant. It is only the sequences of transitions that are shown, not their exact timing. If more than one object of the same class participates, a separate column is assigned to each object. By scanning a particular column in the trace, I can see the transitions that directly affect a particular object. These transitions appear in the **axioms** portion of the formal requirements specification for the object. Figure 5.6 shows state transitions in the formal requirements specification for using a telephone line. This specification contains transitions affecting the phone line, that is, class *phone line*. Figure 5.7 shows an transition trace for a phone call. Note that concurrent transitions can be sent (*Phone line* sends transitions to *Caller* and *Callee* concurrently) and transitions between objects need not alternate (*Caller* dials several digits in succession).

Class *Phone line*

$T_1(S_{Phone_1} \times caller\_lifts\_receiver(Caller) \longmapsto S_{Phone_2} \times dial\_tone\_begins(Caller))$
$T_2(S_{Phone_2} \times dials(3)(Caller) \longmapsto S_{Phone_3} \times dial\_tone\_ends(Caller))$
$T_3(S_{Phone_3} \times dials(9)(Caller) \longmapsto S_{Phone_4})$
$T_4(S_{Phone_4} \times dials(2)(Caller) \longmapsto S_{Phone_5})$
$T_5(S_{Phone_5} \times dials(1)(Caller) \longmapsto S_{Phone_6})$
$T_6(S_{Phone_6} \times dials(2)(Caller) \longmapsto S_{Phone_7})$
$T_7(S_{Phone_7} \times dials(8)(Caller) \longmapsto S_{Phone_8})$
$T_8(S_{Phone_8} \times dials(4)(Caller)$
$\longmapsto S_{Phone_9} \times ringing\_tone(Caller) \times phone\_rings(Callee))$
$T_9(S_{Phone_9} \times answers\_phone(Callee)$
$\longmapsto S_{Phone_{10}} \times tone\_stops(Caller) \times ringing\_stops(Callee))$
$T_{10}(S_{Phone_{10}} \longmapsto S_{Phone_{11}} \times phones\_connected(Caller) \times phone\_connected(Callee))$
$T_{11}(S_{Phone_{11}} \times callee\_hangs\_up(Callee)$
$\longmapsto S_{Phone_{12}} \times connection\_broken(Caller) \times connection\_broken(Callee))$
$T_{12}(S_{Phone_{12}} \times caller\_hangs\_up(Caller) \longmapsto S_{Phone_{13}})$

Class *Caller*

$T_1(S_{Caller_1} \longmapsto S_{Caller_2} \times caller\_lifts\_receiver(Phone\_line))$
$T_2(S_{Caller_2} \times dial\_tone\_begins(Phone\_line) \longmapsto S_{Caller_3} \times dials(3)(Phone\_line))$
$T_3(S_{Caller_3} \times dial\_tone\_ends(Phone\_line) \longmapsto S_{Caller_4} \times dials(9)(Phone\_line))$
$T_4(S_{Caller_4} \longmapsto S_{Caller_5} \times dials(2)(Phone\_line))$
$T_5(S_{Caller_5} \longmapsto S_{Caller_6} \times dials(1)(Phone\_line))$
$T_6(S_{Caller_6} \longmapsto S_{Caller_7} \times dials(2)(Phone\_line))$
$T_7(S_{Caller_7} \longmapsto S_{Caller_8} \times dials(8)(Phone\_line))$
$T_8(S_{Caller_8} \longmapsto S_{Caller_9} \times dials(4)(Phone\_line))$
$T_9(S_{Caller_9} \times ringing\_tone(Phone\_line) \longmapsto S_{Caller_{10}})$
$T_{10}(S_{Caller_{10}} \times tone\_stops(phone\_line) \longmapsto S_{Caller_{11}})$
$T_{11}(S_{Caller_{11}} \times phone\_connected(Phone\_line) \longmapsto S_{Caller_{12}})$
$T_{12}(S_{Caller_{12}} \times connection\_broken(Phone\_line)$
$\longmapsto S_{Caller_{13}} \times caller\_hangs\_up(Phone\_line))$

Class *Callee*

$T_1(S_{Callee_1} \times phone\_rings(Phone\_line) \longmapsto S_{Callee_2} \times answers\_phone(Phone\_line))$
$T_2(S_{Callee_2} \times ringing\_stops(Phone\_line) \longmapsto S_{Callee_3})$
$T_3(S_{Callee_3} \times phones\_connected(Phone\_line)$
$\longmapsto S_{Callee_4} \times callee\_hangs\_up(Phone\_line))$
$T_4(S_{Callee_4} \times connection\_broken(Phone\_line) \longmapsto S_{Callee_5})$

Figure 5.6. State transition specification for phone call.

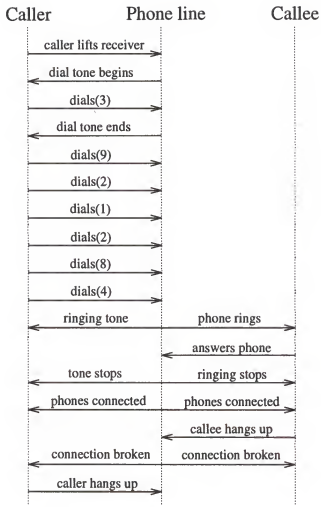

Figure 5.7. Transition trace for phone call.

5.3.2 Development of the Transition Trace Table

I built a transition trace table from the state transitions in the **axioms** portion of the formal requirements specification. Each transition transmits information from one object to another. For example, *dial tone begins* transmits a signal from the phone line to the caller. As I explained before, a state transition consists of current state, input event and object which invokes the input event, next state, and output events and objects in which methods are invoked.

The transition trace table can be constructed as follows:

1. Identify the relevant objects. Each object is represented as a vertical line. For example, there are three vertical lines for the objects *Caller*, *Phone line*, and *Callee* for a phone call system as shown in Figure 5.7.

2. Identify initial transition. I need to identify an initial transition among the first transitions of all objects. A start transition is a transition which has no input event because a transition with an input event entails that it has been triggered by an another object. After identifying an initial transition, draw a horizontal arrow from an object which has an initial transition to the objects which appear in the output events. Next, I move to an object which has an output method and take the following step. For example, the first transition of object *Caller* among three first transitions of objects *Phone_line*, *Caller*, and *Callee* is an initial transition of a phone call system. Next, I move to an object *Phone_line* because it is an object which has output method in the initial transition.

3. Make a transition trace. I draw horizontal arrows from an object which appears with an input event in the formal requirements specification to a current object and from a current object to objects which are appeared with output events. When there is more than one output events, I draw horizontal arrows from a current object to objects which appear with output events.

   When a transition does not have an output event, I handle the next transition in the same object. When the next transition in the same object does not have an input event, I move to the next transition instead of moving an object which has an invoked output method. For example, *caller lifts receiver* is a transition from *Caller* to *Phone_line* and *dial tone begins* is a transition from *Phone_line* to *Caller*. Output events, *ringing tone* to class *Caller* and *phone*

rings to class *Callee*, are concurrent transitions and I draw two horizontal lines, from *Phone line* to *Caller* and from *Phone line* to *Callee*, simultaneously as shown in Figure 5.7.

4. Determine the terminal transition. Among the last transitions of each object, the final transition is that transition which does not have an output event. For example, the last transitions of classes *Phone line* and *Callee* do not have an output event. Hence, these two transitions can be a terminal transition.

CHAPTER 6
CHECKING THE COMPLETENESS AND CONSISTENCY

In this chapter, I will discuss how to apply the top-down and bottom-up approaches to verify the completeness and consistency between the given requirements statements and the OORS. The OORS is represented as graphical notations, such as an inheritance graph, an information tree and a transition trace table. The information tree has two kinds of information, information representing operations inside a class and information representing relationships between classes.

6.1 Top-down Approach

The top-down approach proceeds from the original requirements statements and checks the consistency of each requirements statement with the information in the graphical notations.

The first step in developing anything is to state the requirements. The requirements statements should state what is to be done, not how it is to be done. It should be a statement of needs, not a proposal for a solution. A user manual for the desired system is a good requirements statements. The requester indicates which features are mandatory and which are optional, to avoid overly constraining design decisions. The requester should avoid describing system internals, as this restricts implementation flexibility. Performance specifications and protocols for interaction with external systems are legitimate requirements. Software engineering standards, such as modular construction, design for testability and provision for future extensions are also proper. Many requirements statements, from individuals, companies and government agencies, mix true requirements with design decisions. While there

may sometimes be a compelling reason to require a particular computer or language, there is rarely justification to specify the use of a particular algorithm.

Requirements statements may display a wide range of detail depending on the application. Requirements for a conventional product, such as a payroll program or a billing system, may have considerable detail. Requirements for a research effort in a new area may lack many details, but presumably the research has some objective, which should be clearly stated.

Most requirements statements are ambiguous, incomplete, or even inconsistent. Some requirements are just plain wrong. Some requirements, although precisely stated, have unpleasant consequences on the system behavior or impose unreasonable implementation costs. Some requirements seem reasonable at first but, at some later stage, do not work out as well as one would have thought. The requirements statements are just a starting point for understanding the problem, not an immutable document. The purpose of the subsequent analysis is to fully understand the problem and its implications. There is no reason to expect that requirements statements prepared without a full analysis will be correct. Hence, there is a gap between requirements statements and analysis (that is, requirements specification).

First, I can identify relevant object classes from the application domain to check the completeness and consistency between requirements statements and the result of analysis, that is, the OORS. Objects include physical entities, such as houses, employees and machines, as well as concepts, such as trajectories, seating assignments and payment schedules. All classes must make sense in the application domain. Computer implementation constructs, such as linked lists and subroutines, should be avoided. Not all classes are explicit in the requirements statement; some are implicit in the application domain or general knowledge. Classes often correspond to nouns.

Any dependency between two or more classes is an association. An association can also be a reference from one class to another. Associations often correspond to stative verbs or verb phrases. These include physical location (e.g., next to, part of, contained in), directed actions (e.g., drives), communication (e.g., talks to), ownership (e.g., has, part of), or satisfaction of some condition (e.g., works for, married to, manages). Aggregation is just an association with extra connotations. Most associations between three or more classes can be decomposed into binary associations or phrased as qualified associations. For example, *Cashier enters transaction for account* can be broken into *Cashier enters transaction* and *Transaction concerns account. Bank computer processes transaction against account* can be broken similarly. *ATMs communicate with central computer about transaction* is really the binary associations *ATMs communicate with central computer* and *Transaction entered on ATM.*

Attributes are properties of individual objects, such as name, weight, velocity or color. Attributes are used to show relationships between objects. They should not be mistaken for the objects themselves. Attributes usually correspond to nouns followed by possessive phrases, such as "the color of the car" or "the position of the cursor." Adjectives often represent specific enumerated attribute values, such as *red, on* or *expired.* Unlike classes and associations, attributes are less likely to be fully described in the requirements statements. We have drawn on our knowledge of the application domain and the real world to find them.

For each statement in the original requirements statements, I take the following steps.

1. Specify the statement as the relationship between objects, properties of an object, an operation inside an object or time-dependent behaviors of a system.

2. Find the component in the graphical notations corresponding to each requirements specification.

   - When it is a relationship between objects, such as association, aggregation, etc., I search for the corresponding relationship represented by the dashed lines between the class nodes in the information tree depending on the description of the given requirements statement in the previous step.
   - When it is a property of an object, I find the object node which is specified in the given requirement statement and search for the attribute node which is a child node of the object.
   - When it represents an operation statement, I find a path starting from the root of the information tree and search for the corresponding class, method, attribute and constraint if they exist.
   - When it is a time-dependent dynamic behavior of a system, I search for the corresponding events in the transition trace table which represent the semantics of the given requirements statement.

3. Check for consistency and completeness. I compare the given requirements statement with the information represented by its corresponding path, event transitions, specific node, or dashed lines between the class nodes representing the relationship for consistency checking. If I do not find the corresponding path in the information tree, then I conclude that there is a missing function in the OORS. If I do not find the corresponding relationship between the classes in the information tree, then I conclude that there is a missing relationship between the classes in the OORS. If I do not find the specific attribute node

in the information tree, then I recognize that I missed the property of the object during the requirements analysis phase. If I do not find the sequence of transition events in the transition trace table, then I conclude that I missed a part of the dynamic behaviors of the system during analysis. Also, if I find a path or relationship in the graphical notations, but it is not equivalent to the given statement, then I identify the discrepancies between the two.

I also verify the completeness of the OORS by repeating the above steps for all statements in the requirements statements.

6.2 Bottom-up Approach

The bottom-up approach begins with the utilization of the graphical notations. As was presented in the previous chapter, the graphical notations, such as the inheritance graph, the information tree, and the transition trace table, are constructed after producing the formal specification.

Inheritance is a powerful abstraction for sharing similarities between the classes while preserving their differences. Specifically, multiple inheritance may be used to increase sharing, but only if necessary, because it increases both conceptual and implementation complexity. In using multiple inheritance, it is often possible to designate a primary superclass which supplies most of the inherited structure and behavior. However, the inheritance is a result of an analysis and does not usually appear in the requirements statements. Hence, it is difficult for us to check the consistency between requirements statements and requirements specification.

An information tree captures the static structure of a system by showing the objects in the system, relationship between the objects, and the attributes that characterize each class of objects. It provides an intuitive graphic representation of

a system and is valuable for comparing information with the original requirements statements.

The transition trace table shows the time-dependent behavior of the system and the objects in it. It is important for interactive systems. Sometimes the requirements statements describe the full interaction sequence, but most of the time the interaction format is created by an analyst. For example, the ATM requirements statements indicate the need to obtain transaction data from the user, but it is vague about exactly what parameters are needed and what order to ask for them. The requirements statements may specify needed information but leave open the manner in which it is obtained. I use domain knowledge in the application to describe this time-dependent interactive behavior during the requirements analysis. Therefore, a comparison of information in the transition trace table with the list of domain knowledge is critical for uncovering any inconsistencies.

For each item of information in the graphical notations, I check its consistency comparing the item with the original requirements statements and domain knowledge. The following steps are employed in that process:

1. Construct the statement or the enumeration of the words representing information in the graphical notations.

    - I identify inheritance information in the inheritance graph.
    - I identify information, such as a path from the root to a leaf node, an attribute node representing properties of an object, or a relationship between classes in the information tree. I make the statement or the enumeration of the words representing each path, node, or relationship between the classes in the information tree. Each node represents a specific aspect of the software system I am developing. For example,

when I construct the statement for a path from the root to a leaf node, the root node represents the system, the class node could be a subject, the method could be a verb, the attribute a noun, and the constraint could be an adjective or adverb. However, sometimes I cannot make the statement in English, but I may still make an enumeration of the words from the class name, method name, attribute name and constraint name. A relationship statement between classes represents a physical or conceptual connection between classes. For example, a person *works-for* a company, an ATM *communicates-with* a bank.

- I identify transition events between objects in the transition trace table. I make statements representing these transition events. There are four kinds of typical structures in the transition events shown in Figure 6.1. The transitions of Figure 6.1 show the input and output event transitions of the object *Object*. In case (a), I translate these transitions into "$Object_1$ does $Event_I$ to *Object* and then *Object* does $Event_O$ to $Object_1$." In case (b), I translate it into "$Object_1$ does $Event_I$ to *Object* and then *Object* does $Event_O$ to $Object_2$." In case (c), I translate it into the following statements: "$Object_1$ does $Event_{I_1}$, $Event_{I_2}$, $\cdots$ to *Object*." In case (d), I translate these event transitions into "*Object* does $Event_{O_1}$, $Event_{O_2}$, $\cdots$ to $Object_1$."

2. Check for consistency and completeness. I search for a statement in the original requirements statements corresponding to the statement made in the previous step. If I do not find the equivalent statement in the requirements

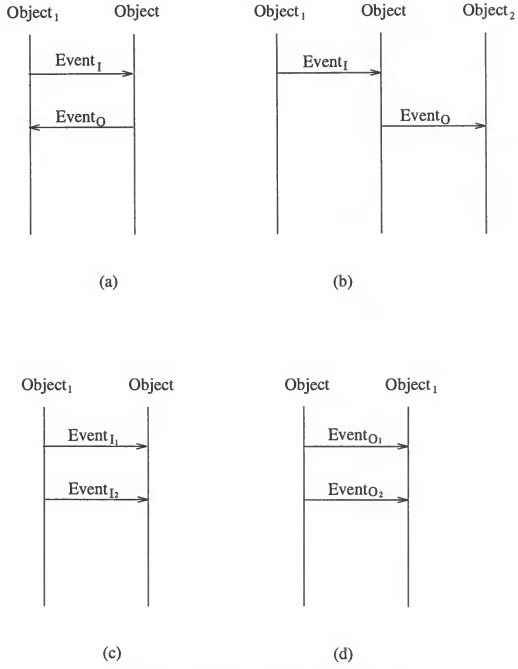

Figure 6.1. Typical structures of transitions.

statements, I determine whether this information is from the domain knowledge by checking against the domain knowledge added during OOA. If this information is not found, I identify an inconsistency in the OORS.

I repeat the above steps for all information, such as paths from root to leaf node in the information tree, relationships between classes from the information tree, and the dynamic behavior of a system from the transition trace table, to verify the completeness of the OORS.

CHAPTER 7
EXAMPLE

In this chapter, I will use a simplified automated teller machine (ATM) system as an example to illustrate our verification approach. Our approach starts from the given requirements statements and the OORS expressed in terms of the object model and dynamic model generated by OOA.

The following requirements statements for an ATM network shown in Figure 7.1 serves as an example throughout the chapter:

> *Develop the software to support a computerized banking system with both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank has its own computer to maintain its accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer of a consortium. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central computer to process transactions, dispenses cash, and prints receipts. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computers.*

The object model shows the static structure of the anticipated software system and organizes it into workable pieces. It also describes the object classes and the relationships between them. The object diagram of the ATM system is shown in Figure 7.2. The dynamic model shows the time-dependent behavior of the system

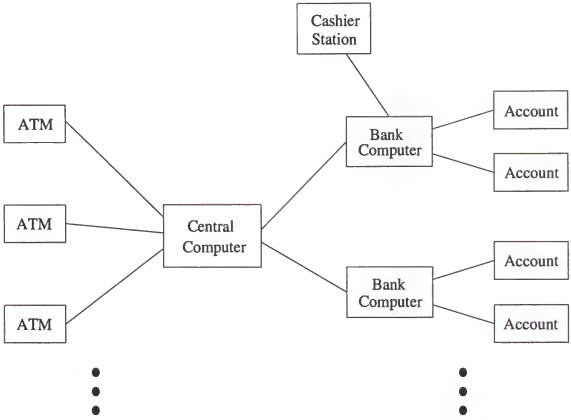

Figure 7.1. ATM system.

and its objects. It is implemented by preparing a state diagram for each object class with dynamic behavior showing the events the object receives and sends. For example, the state transition diagram of the class *ATM* is shown in Figure 7.3. Once I have the object model and dynamic model, I can show how to apply our verification approach by the following steps.

7.1 Producing Formal Specification from the OORS

The first step of our verification approach is to transform the OORS into the corresponding formal object-oriented requirements specification. The following process shows the transformation of the OORS expressed in terms of object and dynamic models into our formal object-oriented requirements specification.

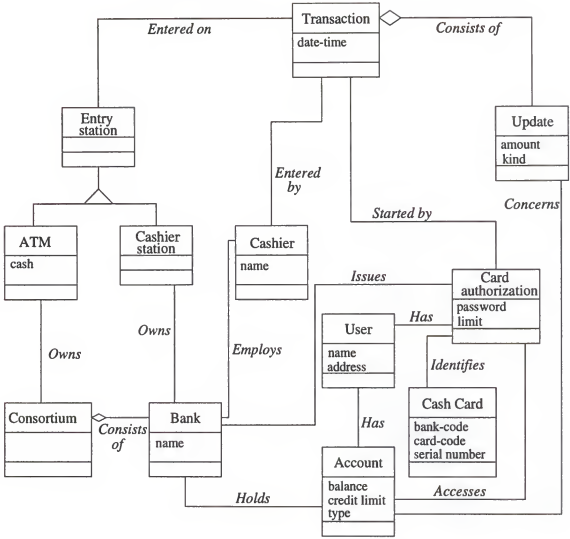

Figure 7.2. The object diagram for the ATM system.

Identify All Objects

All objects in the object model are identified. For instance, in this particular model, the object classes are *ATM*, *Consortium*, *Bank*, *User*, *Account*, *Bank Card*, *Card Authorization*, *Update*, *Transaction*, *Cashier*, and *Cashier station*. Notice that there is no ambiguity in the class name because each one is unique.

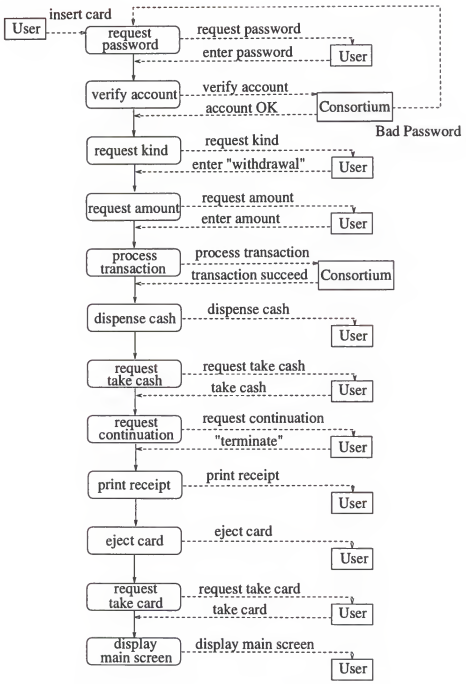

Figure 7.3. The state diagram for class *ATM*.

Specify Attributes of a Class and Their Types

The attributes of a class in an object model become the attributes of the formal object class. For instance, an example of an *ATM* class attribute is *cash*. The attributes of the *Transaction* class are *kind*, *date-time*, and *amount*. The formal specification of a class *ATM* and its attributes is represented as follows:

class spec ATM **is**
  attributes
    cash : *dollars*
    ...
end class spec

In this example, the attribute *cash* can be dollars, pounds, marks, yen, and others. The *cash* itself can have multiple interpretations. But, I put *dollars* as a cash type. That solved an ambiguity problem in the attribute *cash* of a class *ATM*.

Identify Association Relationships

An association is a means for establishing relationships between object classes. For example, the class *ATM* and the class *Consortium* has an association relationship, *Owns*. I create formal specifications for the classes: *ATM*, *Consortium*, and *Ownership*.

class spec *ATM* **is**
  attributes
    Owns : *Ownership*
    attributes of *ATM*
  ...
end class spec

class spec *Consortium* **is**

attributes

Owns : *Ownership*

attributes of *Consortium*

...

end class spec

class spec *Ownership* **is**

attributes

Owns : *ATM*, *Consortium*

attributes of *Ownership*

...

end class spec

If the type of attribute is a class name, it represents the relationship between classes. There is no ambiguity in this example because the classes *ATM* and *Consortium* have an association *Owns* and there is an *Ownership* relationship between them.

Identify Aggregation Relationships

I identify aggregation relationships between classes. Aggregation is the "part-whole" or "a-part-of" relationship. Aggregations are modeled via class composition in our formal object-oriented specification language. For example, a class *Consortium* has an aggregation relationship, *Consists of*, with the class *Bank*, indicating that the object *Consortium* consists of a number of *Bank* objects. The formal specification is as follows:

class spec *Consortium* **is**

attributes

Consists_of : *Bank*

other attributes of *Consortium*

...

end class spec

Specify Constraints of Attributes

A constraint restricts the values that entities can assume. Constraints are modeled by predicates which impose limits on the formal items. For example, the amount of dispensed cash is restricted "less than 200 dollars." Hence, I put the constraint in the attribute *cash* of a class *ATM* as follows:

class spec *ATM* **is**

attributes

cash : *dollars* ($\leq$ 200)

other attributes of *ATM*

...

end class spec

The constraint, $\leq$ 200, is measurable. Hence, no ambiguity in the constraint of the attribute *cash* exists.

Specify Inheritance

Inheritance is modeled at the **inherits** portion in the formal requirements specification. In this ATM example, the class *Entry station* has two subclasses, *ATM* and

Cashier station. That is, the class *ATM* and the class *Cashier station* have a superclass, *Entry station*, at the **inherits** portion of the formal requirements specification of each class.

Transforming Methods into Formal Specifications

A method is a function or transformation that may be applied to or by objects in a class. Each method has a target object as an implicit argument. A method consists of a current class state variable, input variables, a modified class state variable, and output variables. The input and output variables come from the dynamic model. For instance, in this ATM example the input variable of a method *request_password* is a *card* and the class state variable is a *ATM*. The class state variable (ATM) is modified into the new class state variable, ATM', after the method *request_password* is invoked. The **create** method is also involved in instantiating an object. The following is the formal object-oriented requirements specification after the **inherits**, **attributes**, and **methods** portions of the class *ATM* are formed.

class spec *ATM* **is**

inherits

Entry_station

attributes

$cash : dollars\ (\leq 200)$

$Owns : Ownership$

methods

$request\_password(ATM \times card \rightarrow ATM')$

$verify\_account(ATM \times password \rightarrow ATM')$

$request\_kind(ATM \rightarrow ATM')$

$request\_amount(ATM \times kind \rightarrow ATM')$

$process\_transaction(ATM \times amount \rightarrow ATM' \times transaction\_success)$

$dispense\_cash(ATM \times transaction\_success \rightarrow ATM' \times cash)$

$request\_take\_cash(ATM \rightarrow ATM')$

$request\_continuation(ATM \times take\_cash \rightarrow ATM')$

$print\_receipt(ATM \times terminate \rightarrow ATM')$

$eject\_card(ATM \rightarrow ATM' \times card)$

$display\_mainscreen(ATM \rightarrow ATM')$

$create(\rightarrow ATM)$

end class spec

Formalizing Dynamic Model into State Transitions

After a formal requirements specification of an object model, including the classes, their inheritance, attributes, relationships, and methods, is formed, the next step is to specify how the state may change according to the dynamic model. The **axioms** portions of the formal requirements specification represent the state transitions axiomatically. That is, modeling a state transition in the formal requirements specification simply means specifying a schema for every state transition. A schema consists of a state being altered by the transition, an input event that causes the transition, the object which requests the input event, the state after a transition, and output events along with their destination objects. For example, in the formal requirements specification of the class *ATM* the state being altered by an input event *insert_card* is S_{ATM_1} and the state after the transition is S_{ATM_2}. The output event *request_password* of the transition T_1 in the class *ATM* becomes an input event to the class *User*. The following is the complete formal object-oriented requirements specification of a class *ATM*:

class spec ATM **is**

inherits

Entry_station

attributes

$cash : dollars\ (\leq 200)$

$Owns : Ownership$

methods

$request\_password(ATM \times card \rightarrow ATM')$

$verify\_account(ATM \times password \rightarrow ATM')$

$request\_kind(ATM \rightarrow ATM')$

$request\_amount(ATM \times kind \rightarrow ATM')$

$process\_transaction(ATM \times amount \rightarrow ATM' \times transaction\_success)$

$dispense\_cash(ATM \times transaction\_success \rightarrow ATM' \times cash)$

$request\_take\_cash(ATM \rightarrow ATM')$

$request\_continuation(ATM \times take\_cash \rightarrow ATM')$

$print\_receipt(ATM \times terminate \rightarrow ATM')$

$eject\_card(ATM \rightarrow ATM' \times card)$

$display\_mainscreen(ATM \rightarrow ATM')$

$create(\rightarrow ATM)$

axioms

select

$T_1(S_{ATM_1} \times insert\_card(User) \longmapsto S_{ATM_2} \times request\_password(User))$

$T_1(S_{ATM_1} \times bad\_password(Consortium) \longmapsto S_{ATM_2} \times request\_password(User))$

end select

$T_2(S_{ATM_2} \times enter\_password(User) \longmapsto S_{ATM_3} \times verify\_account(Consortium))$

$T_3(S_{ATM_3} \times account\_OK(Consortium) \longmapsto S_{ATM_4} \times request\_kind(User))$

$T_1(S_{User_1} \longmapsto S_{User_2} \times insert\_card(ATM))$
$T_2(S_{User_2} \times request\_password(ATM) \longmapsto S_{User_3} \times enter\_password(ATM))$
$T_3(S_{User_3} \times request\_kind(ATM) \longmapsto S_{User_4} \times enter\_kind(ATM))$
$T_4(S_{User_4} \times request\_amount(ATM) \longmapsto S_{User_5} \times enter\_amount(ATM))$
$T_5(S_{User_5} \times request\_take\_cash(ATM) \longmapsto S_{User_6} \times take\_cash(ATM))$
$T_6(S_{User_6} \times request\_continuation(ATM) \longmapsto S_{User_7} \times terminate(ATM))$
$T_7(S_{User_7} \times request\_take\_card(ATM) \longmapsto S_{User_8} \times take\_card(ATM))$
$T_8(S_{User_8} \times display\_main\_screen(ATM) \longmapsto S_{User_9})$

Figure 7.4. The **axioms** portion of a class *User*.

$T_4(S_{ATM_4} \times enter\_kind(User) \longmapsto S_{ATM_5} \times request\_amount(User))$

$T_5(S_{ATM_5} \times enter\_amount(User) \longmapsto S_{ATM_6} \times process\_transaction(Consortium))$

$T_6(S_{ATM_6} \times transaction\_succeed(Consortium) \longmapsto S_{ATM_7} \times dispense\_cash(User))$

$T_7(S_{ATM_7} \longmapsto S_{ATM_8} \times request\_take\_cash(User))$

$T_8(S_{ATM_8} \times take\_cash(User) \longmapsto S_{ATM_9} \times request\_continuation(User))$

$T_9(S_{ATM_9} \times terminate(User) \longmapsto S_{ATM_{10}} \times print\_receipt(User))$

$T_{10}(S_{ATM_{10}} \longmapsto S_{ATM_{11}} \times eject\_card(User))$

$T_{11}(S_{ATM_{11}} \longmapsto S_{ATM_{12}} \times request\_take\_card(User))$

$T_{12}(S_{ATM_{12}} \times take\_card(User) \longmapsto S_{ATM_{13}} \times display\_main\_screen(User))$

end class spec

Each $S_{ATM_i}, i = 1, 2, \cdots$, represents a state in the dynamic model of the class *ATM*. That is, S_{ATM_1} is a state *request password*, S_{ATM_2} is a state *verify account*, S_{ATM_3} is a state *request kind*, $\cdots$, etc. The state transition specification of the classes *User*, *Consortium*, and *Bank* are shown in Figure 7.4, Figure 7.5, and Figure 7.6, respectively.

$$
\begin{aligned}
&T_1(S_{Consortium_1} \times verify\_account(ATM) \longmapsto S_{Consortium_2} \times verify\_card(Bank)) \\
&T_2(S_{Consortium_2} \times bank\_account\_OK(Bank) \\
&\qquad \longmapsto S_{Consortium_3} \times account\_OK(ATM)) \\
&T_3(S_{Consortium_3} \times process\_transaction(ATM) \\
&\qquad \longmapsto S_{Consortium_4} \times process\_bank\_transaction(Bank)) \\
&T_4(S_{Consortium_4} \times bank\_transaction\_succeed(Bank) \\
&\qquad \longmapsto S_{Consortium_5} \times transaction\_succeed(ATM))
\end{aligned}
$$

Figure 7.5. The **axioms** portion of a class *Consortium*.

$$
\begin{aligned}
&T_1(S_{Bank_1} \times verify\_card(Consortium) \\
&\qquad \longmapsto S_{Bank_2} \times bank\_account\_OK(Consortium)) \\
&T_2(S_{Bank_2} \times process\_bank\_transaction(Consortium) \\
&\qquad \longmapsto S_{Bank_3} \times bank\_transaction\_succeed(Consortium))
\end{aligned}
$$

Figure 7.6. The **axioms** portion of a class *Bank*.

7.2 Building the Graphical Notations

The Inheritance Graph

The inheritance graph represents the hierarchical structures among classes. First of all, I identify all classes from the formal requirements specification, such as *User*, *ATM*, *Bank*, *Consortium*, $\cdots$, etc., and make a node for each class in an inheritance graph. In order to construct a hierarchical graph, the **inherits** portion of each formal requirements specification is checked for every class. In this ATM example, an *ATM* class has a superclass *Entry station* and a *Cashier station* class has a superclass *Entry station*. That is, a *Entry station* class has two children *ATM* and *Cashier station* in the inheritance graph. Now, I make a connection from root node which represents the software system to each class node which does not have parent nodes. Figure 7.7 shows the inheritance graph of the ATM system.

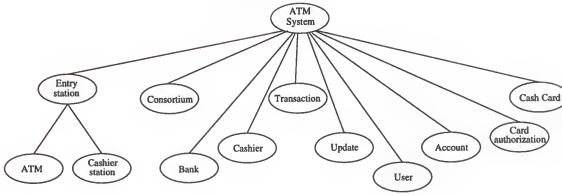

Figure 7.7. The inheritance graph of an ATM system.

The Information Tree

Next, I build an information tree from the formal requirements specification. An information tree is a tree which describes the static characteristics of classes and the relationships between classes. It represents the classes, associations and aggregations between classes, as well as the methods and attributes of each class. When I build the information tree, I add information, such as object classes, methods for each class, attributes of a method, and constraints for attributes as sequences. In the ATM example I first identify classes and their relationships. I have classes, such as *User*, *ATM*, *Bank*, *Consortium*, $\cdots$, etc., and I can identify the relationships between the classes. For example, the class *ATM* invokes methods in the class *User* and conversely, the class *User* invokes methods in the class *ATM*. The classes *ATM* and *Consortium* have an association relationship, *Ownership*, to each other and the class *Consortium* has an aggregation relationship, *Consists of*, to the class *Bank*. The information tree after the inclusion of information for the object classes and their relationships is shown in Figure 7.8. Next, I identify the methods for each class. It comes from the **methods** portions of the formal requirements specification. For example, the class *ATM* has methods, such as *request_password*, *request_amount*, *dispense_cash*, *create*, $\cdots$, etc. The information tree after including the information

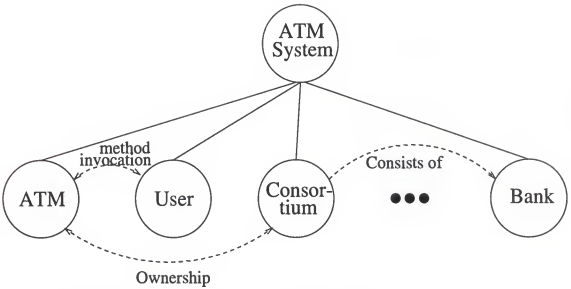

Figure 7.8. The information tree after adding class information.

for the methods in each class is shown in Figure 7.9. I identify the attributes of each method from the **attributes** portion in the formal requirements specification. If the attribute is an input parameter of a method, I put a letter "D" on the edge between the method and the attribute. Similarly, if the attribute is an output parameter of a method, then I put a "R" between them. For example, the method *request_password* in the class *ATM* has an attribute, *cash*, as an input parameter and the method *dispense_cash* has an input parameter *transaction_success* and an output parameter *cash*. As another example, let us examine the method *create*. In the class *ATM*, the method *create* has an attribute *cash* because it is an attribute of the class *ATM*. I finally identify any constraints there might be for the attributes. This comes from the **attributes** portion in the formal requirements specification. For example, the attribute *cash* has an constraint "less than 200 dollars." Hence, all *cash* nodes in the class *ATM* have a child node (≤ 200) as a constraint. That is, I add the information for the attributes of each method and constraints for the attributes if there are any as shown in Figure 7.10.

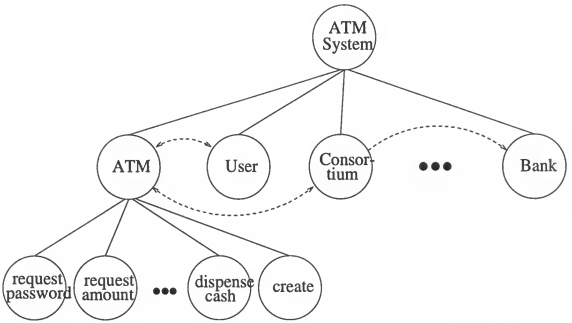

Figure 7.9. The information tree after identifying methods in each class.

The Transition Trace Table

The transition trace table represents an ordered list of transitions between different objects. I built a transition trace table from the state transitions in the **axioms** portion of the formal requirements specification. Each transition transmits information from one object to another. First, I identify objects. Each object is represented as a vertical line. For example, when I describe a transition trace table for only four objects *User*, *ATM*, *Consortium*, and *Bank*, there are four vertical lines for the objects of an ATM system as shown in Figure 7.11. Second, I need to identify a start transition. The start transition is that transition among the first in each class with no input event. For example, the first transition, *insert_card*, of the object *User* is an initial transition. I draw horizontal arrows from an object which appears with an input event in a formal requirements specification to a current object and from a current object to objects which appear with output events. Finally, I decide the terminal transition. The final transition is that transition among the

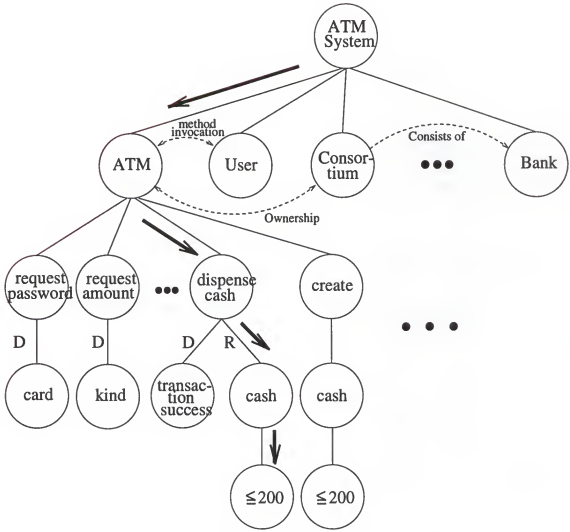

Figure 7.10. The information tree for the ATM system.

last in each class with no output event. For example, the last transition of the ATM system with four objects, such as *User*, *ATM*, *Consortium*, and *Bank*, is a transition *display_main_screen* from the object *ATM* to *User*. The transition trace table for the ATM system with four objects is shown in Figure 7.11.

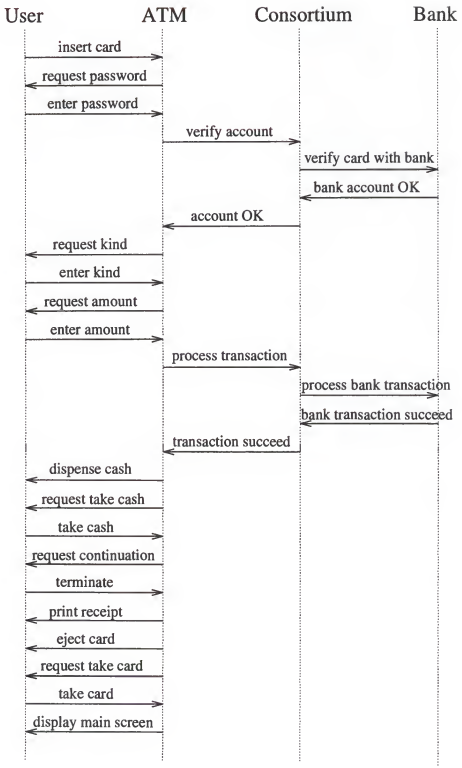

Figure 7.11. Transition trace table for the ATM system.

7.3 Checking the Completeness and Consistency

7.3.1 Top-down Approach

For each statement in the given requirements statements, I apply the top-down approach. Consider the example: *ATM dispenses cash.*

1. This statement is classified as an operation statement in an object *ATM.*

2. The corresponding path of this statement in the information tree is identified by selecting nodes, such as ATM, dispense_cash, cash, and $\leq$ 200 as shown with arrows in Figure 7.10.

3. I compare this path and the statement for the consistency. The result reveals an inconsistency, $\leq$ 200, in the OORS.

7.3.2 Bottom-up Approach

I apply the bottom-up approach as follows:

Consider the path in the information tree, *ATM*, *dispense_cash*, *cash*, and *less than or equal to $200*, as an example.

1. I make the statement or the enumeration of words for that path as follows: ATM dispenses cash (less than or equal to $200).

2. I search for a statement in the requirements statements that matches with the information represented by that path. I find the statement in original requirements statement, "ATM dispenses cash". I determine that *less than or equal to $200* is from the domain knowledge or a mistake made during the OOA. Comparing it with the domain knowledge added during the OOA, I find that "less than or equal to $200" is from the domain knowledge.

CHAPTER 8
DISCUSSIONS

I have developed an effective verification approach to the requirements specification generated during OOA. In this dissertation, I have presented (a) the formal object-oriented requirements specification language and its transformation procedure from the object-oriented requirements specification resulting from analysis, (b) the graphical notations for user-friendly representations and their transformation procedures, and (c) the approach for checking the consistency and completeness between the object-oriented requirements specification described by graphical notations and the original requirements statements given by clients.

I have developed a formal object-oriented requirements specification language in which the object-oriented concept is incorporated into a transition based formal specification without sacrificing the advantages of either. The object-oriented requirements specification is given by object and dynamic models. Ambiguities in the requirements statements may lead to different interpretations of the software system, thus making verification more difficult. All ambiguities in the requirements statements in a natural language have been solved during OOA by communicating with the user or using domain knowledge. In the verification, we ensure that the requirements specification has no ambiguity and is complete and consistent with the original requirements statements. One way to avoid the ambiguity inherent in natural language is to write the requirements specification in a formal requirements specification language. To do this, I identify the source of ambiguities in requirements statements and check the testability of the requirements specification (that is, the object and dynamic models) during the transformation of the requirements

specification to formal requirements specification. Testable requirements specification is specific and unambiguous. When I transform the requirements specification to formal specification, I check the testability of each piece of information in the requirements specification to ensure that there are no ambiguities in the requirements specification. Transformation procedures which ensure that there is an exact translation from the object-oriented requirements specification to formal requirements specification have also been developed.

I have developed graphical notations for representing the object-oriented requirements specification pictorially. By representing the object-oriented requirements specification as graphical notations, it becomes easy for the software developer to understand and check the requirements specification, and it is simpler to visualize the structure of the software system under development. I have made three different graphical representations from the formal requirements specification: the inheritance graph, the information tree and the transition trace table. The inheritance graph contains only classes and represents the inheritance relationships among classes. The notation used for inheritance is a tree structure. Strictly speaking, it's not a tree, but a graph designed to support multiple inheritance because our object-oriented software development methodology supports multiple inheritance. An information tree is a tree which describes the static characteristics of classes and relationships between those classes. It represents the classes, associations and aggregations between the classes, as well as the methods and attributes of each class. I have developed a procedure for building graphical notations from the formal requirements specification. I can explore information in the object-oriented requirements specification by traversing the information tree. However, the notations used in the information tree do not represent all dynamic behavior, most of which comes from the domain knowledge. I use a transition trace table to represent the dynamic

behaviors of a system. The transition trace is an ordered list of transitions between different objects assigned to columns in a table. By scanning a particular column in the trace, we can see the transitions that directly affect a particular object.

I have developed a two-way verification approach in which top-down and bottom-up approaches are used. The top-down approach proceeds from the original requirements statements and checks the consistency of each requirements statement with the information in the graphical notations. I specify the original requirements statement as the relationship between objects, properties of an object, an operation inside an object, or a time-dependent behavior of a system. Then, I find the component in the graphical notations corresponding to each requirements specification. The bottom-up approach begins with the graphical notations. I check the consistency of information in the graphical notations by comparing them with the original requirements statements. I repeat the above steps for all information to verify the completeness of the object-oriented requirements specification.

Considering that many errors in software stem from errors in the requirements statements, our approach can contribute to the reduction of software development cost by identifying the potential problems at the requirements analysis phase of the software development cycle. Our approach can also be easily automated because I can systematically compare the requirements statements with the object-oriented requirements specification through a series of steps so that the results of the requirements verification can be quickly made available to the software developers. The graphical notations are built to organize the information so that requirements verification can be systematically performed without increasing the complexity of the problem.

An important area of future research will be the development of software tools and environments to support our approach. Specially, priority should be given to

the development of means for automatically transforming the OORS into formal specification and for graphically representing the inheritance graph, information tree and transition trace table.

Another important issue in the verification of software development is design verification. In the requirements verification, I verify the completeness and consistency between requirements statements and requirements specification. In the design verification, I extend our approach with more specific design issues, such as concurrency, communication, and deadlock in software development for distributed computing systems. Software design is critical in terms of the quality and reliability of the final product. It may be possible to produce a poor implementation from a good design, but it is seldom possible to produce a good implementation from a poor design.

APPENDIX
FORMAL SPECIFICATION LANGUAGE SYNTAX

class_spec_system → *class_spec* { *class_spec* } *
class_spec → **class spec** *class_spec_id* **is** *class_spec_exp* **end class spec**
class_spec_exp → **inherits** *inherited_classes*
attributes *attributes*
methods *methods*
axioms *axioms*
inherited_classes → *inherited_class* {, *inherited_class* } *
attributes → *attribute attributes*
| *attribute*
methods → *method methods*
| *method*
axioms → *axiom axioms*
| *axiom*
inherited_class → *class_spec_id*
attribute → *attribute_variable* : *attribute_type* { (*constraint*) }
method → *method_name* (*input_types* → *output_types*)
axiom → **select** *Transitions* **end select**
| **if** *relational_exp* **then** *Transition* **else** *Transition*
| **while** *relational_exp Transitions*
| *Transition*
Transitions → *Transition Transitions*
| *Transition*
Transition → **when** (*relational_exp*) transition (*current_state*
× *input_event* (*object_name*) → *modified_state* × *output_events*)
| transition (*current_state* × *input_event* (*object_name*)
→ *modified_state* × *output_events*)
output_events → *output_event* { × *output_event* } *
output_event → *output_event_name* (*object_name*)
attribute_variable → **identifier**
attribute_type → *class_spec_id* | **identifier**
constraint → *relational_exp relational_op relational_exp*
method_name → **identifier**
input_types → *class_spec_id* { × *input_type* } *
input_type → *class_spec_id* | **identifier**
output_types → *class_spec_id* { × *output_type* } *
output_type → *class_spec_id* | **identifier**

current_state → **identifier**
modified_state → **identifier**
input_event → **identifier**
output_event_name → **identifier**
object_name → **identifier**
relational_exp → *relational_exp add_op multi_exp*
| *multi_exp*
multi_exp → *multi_exp multi_op express* | *express*
express → **identifier** | *numbers* |
numbers → *digit* { *digit* } *
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
relational_op → < | ≤ | > | ≥ | = | !=
add_op → + | -
multi_op → * | / | %
class_spec_id → **identifier**

REFERENCES

[1] S. S. Yau, X. Jia, and D.-H. Bae, "Trends in Software Design for Distributed Computing Systems," *Proc. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, Cairo, Egypt, September 1990, pp. 154–160.

[2] S. S. Yau, X. Jia and D.-H. Bae, "Software Design Methods for Distributed Computing Systems," *Journal of Computer Communications*, Vol. 15, No. 5, May 1992, pp. 213–223.

[3] Grady Booch, "Object-Oriented Development," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, Feb. 1986, pp. 211–221.

[4] S. S. Yau, and G. -H. Oh, "An Object-Oriented Approach to Software Development for Autonomous Decentralized Systems," *Proc. International Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, March 1993, pp. 37–43.

[5] S. S. Yau, D.-H. Bae and M. Chidambaram, "Object-Oriented Development of Architecture Transparent Software for Distributed Parallel Systems," Journal of Computer Communication, Vol. 16, No. 5, May 1993, pp. 317–326.

[6] S. S. Yau, K. Yeom, B. Gao, L. Li and D.-H. Bae, "An Object-Oriented Software Development Framework for Autonomous Decentralized Systems," *Proc. International Symposium on Autonomous Decentralized Systems*, Phoenix, AZ, April 1995, pp. 405–411.

[7] Stephen S. Yau, X. Jia, D-H. Bae, M. Chidambaram and G. Oh, "An Object-Oriented Approach to Software Development for Parallel Processing Systems," *Proc. 15th COMPSAC 91*, Tokyo, Japan, September 1991, pp. 453–458.

[8] J. E. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[9] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis*, Yourdon Press, Englewood Cliffs, NJ 1988.

[10] P. Coad and E. Yourdon, *Object-Oriented Analysis,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[11] W. J. Quirk, *Verification and Validation of Real-Time Software*, Springer-Verlag, Berlin, Heidelberg, 1985.

[12] S. S. Yau, D.-H. Bae and K. Yeom, "An Approach to Object-Oriented Requirements Verification in Software Development for Distributed Computing Systems," *Proc. 18th International Computer Software & Applications Conference,* Taipei, Taiwan, November 1994, pp. 96–102.

[13] Grady Booch, "Object-Oriented Design," *ACM Ada Letters*, Vol. 1 No. 3, 1982, pp. 64–76.

[14] Grady Booch, *Object-Oriented Analysis and Design with Applications,* Benjamin/Cummings, Redwood City, California, 1993.

[15] U. S. Department of Defense, *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A, Washington D.C., January 1983.

[16] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[17] M. E. S. Loomis, A. V. Shah and J. E. Rumbaugh, "An Object Modelling Technique for Conceptual Design," *in Proc. 1st European Conference on Object-Oriented Programming (ECOOP '87),* J. Bezivin, J.-M. Hullot, P. Cointe and H. Lieberman (eds.) Lecture Notes in Computer Science, Vol. 276, Springer-Verlag, Berlin, 1987, pp. 192–202.

[18] P. Coad and E. Yourdon, *Object-Oriented Design,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[19] B. Henderson-Sellers, *A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation: A New Approach to Software Engineering,* Prentice-Hall Object-Oriented Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[20] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

[21] A. I. Wasserman, "Cruise Control System and the OOSD/Ada Design Editor," Interactive Development Environments, San Francisco, California, 1989.

[22] A. I. Wasserman, P. A. Pircher and R. J. Muller, "An Object-Oriented Structured Design Method for Code Generation," *ACM Software Engineering Notes,* Vol. 14, No. 1, 1989, pp. 32–55.

[23] A. I. Wasserman, P. A. Pircher and R. J. Muller, "The Object-Oriented Structured Design Notation for Software Design Representation," *IEEE Computer,* Vol. 23, No. 3, 1990, pp. 50–63.

[24] R. J. A. Buhr, G. M. Karam and C. M. Woodside, "Software CAD: A Revolutionary Approach," *IEEE Transactions on Software Engineering,* Vol. 15, No. 3, 1989, pp. 235–249.

[25] E. Yourdon, *Modern Structured Analysis,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[26] T. DeMarco, *Structured Analysis and Systems Specification,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[27] C. P. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques,* Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[28] M. Page-Jones, *The Practical Guide to Structured Systems Design,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[29] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[30] P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems,* Vols 1–3, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1985–86.

[31] P. P. Chen, "The Entity-Relationship Model: Towards a Unified View of Data," *ACM Transactions on Database Systems,* Vol. 1, No. 1, 1976, pp. 9–36.

[32] K. Shumate and M. Keller, *Software Specification and Design: A Disciplined Approach for Real-Time Systems,* John Wiley & Sons, Inc., New York, New York, 1992.

[33] J. R. Cameron (ed.), *JSP and JSD: The Jackson Approach to Software Development,* IEEE Computer Society, Washington D.C., 1989.

[34] M. A. Jackson, *System Development,* Prentice-Hall International Series in Computer Science, Prentice-Hall, London, 1983.

[35] G. Cutts, *Structured Systems Analysis and Design Methodology,* Alfred Waller, Henley-on-Thames, Oxfordshire, UK, 1992.

[36] M. Goodland and C. Ashworth, *SSADM: A Practical Approach,* McGraw-Hill, New York, 1992.

[37] M. Goodland, "Object Orientation: The Emperor's New Clothes ?", *Conference on Object-Oriented Techniques,* Kingston University, UK, 1992.

[38] P. C. Masiero and F. S. R. Germano, "JSD as an Object-Oriented Design Method," *ACM Software Engineering Notes,* Vol. 13, No. 3, 1988, pp. 22–23.

[39] P. T. Ward, "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software,* Vol. 6, No. 3, March 1989, pp. 74–82.

[40] E. V. Seidewitz, "General Object-Oriented Software Development: Background and Experience," *Journal of Systems and Software,* Vol. 9, No. 2, 1989, pp. 95–108.

[41] M. Heitz, "HOOD: A Hierarchical Object-Oriented Design Method," *Proc. 3rd German Ada Users Congress,* Gesellschaft fuer Software Engineering, Munich, Germany, 1988, pp. 12-1–12-9.

[42] R. Di Giovanni and P. L. Iachini, "HOOD and Z for the Development of Complex Software Systems," *VDM'90 VDM and Z – Formal Methods in Software Development*, Springer-Verlag, Berlin, 1990, pp. 262–289.

[43] D. W. Embley, B. D. Kurtz and S. N. Woodfield, *Object-Oriented Systems Analysis:A Model Driven Approach,* Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[44] R. J. Wirfs-Brock, B. Wilkerson and L Wiener, *Designing Object-Oriented Software,* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[45] T. Kondo, M. Inoue, K. Nakai, K. Doi and Y. Suzuki, "Application of Autonomous Decentralized System to the Steel Product Computer Control," *Proc. Workshop on the Future Trends of Distributed Computing Systems in the 1990s, Hong Kong*, April 1992, pp. 419–423.

[46] H. Ihara and K. Mori, "Autonomous Decentralized Computer Control Systems," *IEEE Computer*, Vol. 17, No. 8, 1984, pp. 57–66.

[47] K. Kawano, M. Orimo and K. Mori, "Autonomous decentralized systems: Concept, data field architecture and future trend," *Proc. International Symposium on Autonomous Decentralized Systems (ISADS 93), Kawasaki, Japan,* March 30–April 1 1993, pp. 28–34.

[48] K. Mori, H. Ihara, Y Suzuki, K. Kawano, M. Koizumi, M. Orimo, K. Nakai and H. Nakanish, "Autonomous Decentralized Software Structure and Its Application," *Fall Joint Computer Conference*, Dallas, TX, November 1986, pp. 1056–1063.

[49] M. Orimo, S. Hirasawa, H. Fujise, M. Takeuch and K. Mori, "Autonomous Decentralized File System to and Its Application," *Proc. Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Taipei, Taiwan, April 1992, pp. 262–268.

[50] I. Sommerville, *Software Engineering*, Addison-Wesley Publishing Co., Workingham England, 1992.

[51] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Co., Reading, MA, 1986.

[52] T. Cargill, *C++ Programming Style,* Addison-Wesley, Reading, MA, 1992

[53] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley , Reading, MA, 1990

[54] J. Vlissides, "Generalized Graphical Object Editing," Technical Report: CSL-TR-90-427, Computer Systems Laboratory, Stanford University, Stanford, CA, June 1990

[55] R. W. Scheifer and J. Gettys, "The X Window System," *ACM Trans. on Graphics,* Vol. 5, No. 2, 1986, pp.79-102

[56] Mark A. Linton, Paul R. Calder, John A. Interrante, Steven Tang and John M. Vlissides, "InterViews Reference Manual, Version 3.1," Stanford University, Stanford, CA, 1992

[57] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4, April 1989, pp. 541–580.

[58] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," in *Current Trends in Concurrency*, ed. Bakker, Roever, Rozenberg, Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, 1986, pp. 510–584.

[59] Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York, New York, 1992.

[60] M. Johnson and P. Sanders, "From Z Specifications to Functional Implementations," *Z User Workshop*, ed. John E. Nicholls, Springer-Verlag, Rewley House, Oxford, 1990, pp. 86–112.

[61] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[62] Bernard Berthomieu and Michel Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets," *IEEE Transactions on Software Engineering*, Vol. 17. No. 3, Mar. 1991, pp. 259–273.

[63] Richard Gerber abd Insup Lee, "A Layered Approach to Automating the Verification of Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. 18, No. 9, Dec. 1992, pp. 768–784.

[64] C. J. Fidge, "Specification and Verification of Real-Time Behaviour using Z and RTL," *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1991, pp. 393–410.

[65] D. Coleman, P. Arnold, S. Bodoff, H. Gilchrist, and F. Hayes, "An Evaluation of Five Object-Oriented Development Methods," Hewlett-Packard Technical Report, May 1991.

[66] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, January 1992.

[67] L. M. G. Feijs and H. B. M. Jonkers, *Formal Specification and Design*, Cambridge University Press, Cambridge, UK, 1992.

[68] Alan M. Davis, *Software Requirements Analysis & Specification*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[69] IEEE Std 830-1984, *IEEE Guide to Software Requirements Specifications*, IEEE, New York, 1984.

[70] Donald C. Gause and Gerald M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House Publishing, New York, NY, 1989.

[71] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[72] Barry W. Boehm, Terence E. Gray and Thomas Seewaldt, "Prototyping versus Specifying: A Multiproject Experiment," *IEEE Transactions on Software Engineering*, Vol. 10, No. 3, May 1984, pp. 290–303.

[73] Alan M. Davis, "Rapid Prototyping Using Executable Requirements Specifications," *ACM Software Engineering Notes*, Vol. 7, No. 5, December 1982, pp. 39–44.

[74] H. Gomaa, "The Impact of Rapid Prototyping on Specifying User Requirements," *ACM Software Engineering Notes*, Vol. 8, No. 2, April 1983, pp. 17–28.

[75] R. Balzer, N. Goldman and D. Wile, "Operational Specifications as a Basis for Rapid Prototyping," *ACM Software Engineering Notes*, Vol. 7, No. 5, December 1982, pp. 3–16.

[76] D. McCracken and M. Jackson, "Life Cycle Concept Considered harmful," *ACM Software Engineering Notes*, Vol. 7, No. 2, April 1982, pp. 29–32.

[77] R. Mason and T. Carey, "Prototyping Interactive Information Systems," *Communication of the ACM*, Vol. 26, No. 5, May 1983, pp. 347–354.

[78] M. Zelkowitz, "A Case Study in Rapid Prototyping," *Software Practice and Experience,* Vol. 10, No. 12, December 1980, pp. 1037–1042.

[79] Eduardo Casais, Claus Lewerentz, Thomas Linder, and Franz Weger, "Formal Methods and Object-Orientation," *Tutorial at TOOLS Europe*, Versailles, France, 1993.

[80] Ruth Breu, *Algebraic Specification Techniques in Object Oriented Programming Environment*, LNCS, Springer-Verlag, Berlin, Heidelberg, 1991.

[81] Ravi Sethi, *Programming Languages*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1989.

[82] Henri E. Bal, Jennifer G. Steiner and Andrew S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Survey*, Vol. 21, No. 3, September 1989, pp. 261–322.

[83] Robert O. Lewis, *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*, John Wiley & Sons, New York, 1992.

BIOGRAPHICAL SKETCH

Keunhyuk Yeom was born on February 2, 1962, in Incheon, Korea. He received a B.S. degree in computer science and statistics from Seoul National University in 1985. After graduation, he worked as a researcher at GoldStar Information and Systems R & D Laboratory in Seoul, Korea, from 1985 to 1990. He earned an M.S. degree in computer and information sciences from the University of Florida in 1992. His current research interests are object-oriented software development methodologies, formal requirements specification and verification, real-time software systems, distributed and parallel processing systems and their applications, specially computer integrated manufacturing and management information systems.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate. in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stephen S. Yau, Chair
Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate. in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Paul A. Fishwick
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate. in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Li-Min Fu
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate. in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Andrew F. Laine
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Paul Chun
Professor of Biochemistry and
Molecular Biology

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 1995

Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School